UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

Szenariobasierte Synthese verteilter mechatronischer Systeme

Schriftliche Arbeit

zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)

vorgelegt von

Christian Brenner

Paderborn, September 2015

Zusammenfassung

Mechatronische Systeme werden primär durch Software gesteuert, enthalten jedoch auch mechanische Anteile. Bei Fehlern in der Software drohen deshalb schwere Unfälle. Diese ist daher sicherheitskritisch und muss vollständig korrekt sein. Mechatronische Systeme sind jedoch oft verteilte Systeme, in denen mehrere Subsysteme immer umfangreichere Funktionen durch Kooperation erfüllen müssen. Ihre Korrektheit hängt somit besonders von der komplexen Kommunikation zwischen den Subsystemen ab. Korrektheit bedeutet, dass das System seine Anforderungen erfüllt. Durch formale Spezifikation und Analyse der Anforderungen kann vermieden werden, dass sie ungenau oder inkonsistent sind. Ihre manuelle Implementierung ist aufgrund ihrer Komplexität jedoch fehleranfällig. Manuelle Fehler der Entwickler können durch eine Automatisierung des Wegs von den Anforderungen zur fertigen Software vermieden werden.

In dieser Dissertation wurde ein Verfahren zur *verteilten Synthese* entwickelt, das aus formalen Anforderungen an das Kommunikationsverhalten eines verteilten Systems automatisiert ein Implementierungsmodell ableitet, welches diese Anforderungen erfüllt. Der Hauptvorteil gegenüber vergleichbaren Ansätzen ist, dass automatisch zusätzlich benötigte Nachrichten identifiziert und ergänzt werden, während das Hinzufügen unnötiger Nachrichten vermieden wird. Weiterhin untersucht diese Arbeit die Anwendung der verteilten Synthese speziell zur Erzeugung der Kommunikationssoftware von verteilten mechatronischen Systemen. Hierfür wurde das Verfahren um die Berücksichtigung von Echtzeit-Anforderungen erweitert.

Aufgrund der hohen Komplexität des zu entwickelnden Systemverhaltens können Entwickler dieses nicht vollständig überblicken. Daher wurden in früheren Arbeiten szenariobasierte Spezifikationstechniken entwickelt. Sie ermöglichen den Entwicklern die Konzentration auf konkrete Situationen, in denen jeweils nur wenige Anforderungen relevant sind. Der in dieser Arbeit entwickelte Ansatz zur verteilten Synthese operiert auf szenariobasierten Anforderungen, die mittels Modal Sequence Diagrams spezifiziert werden. Um das durch die Synthese erzeugte Modell zur fertigen Software weiterentwickeln zu können, stellt die Arbeit Konzepte zur Integration des entwickelten Ansatzes in die Entwurfsmethode MechatronicUML vor.

Abstract

Mechatronic systems are mainly controlled by software, but they also contain mechanical components. Errors in the software can lead to devastating accidents and, therefore, need to be prevented. Mechatronic systems are often distributed systems where several subsystems have to cooperate to realize increasingly advanced functions. Thus, the correctness of these systems depends especially on the complex communication among subsystems. Correctness means that the system fulfills its requirements. Formal specification and analysis of requirements can prevent that they are unclear or inconsistent. However, their complexity makes their manual implementation prone to errors. Manual errors by the developers can be prevented by automating the process of deriving the final software from the given requirements.

This dissertation thesis presents a *distributed synthesis* approach that automatically derives an implementation model from a formal requirements specification for the communication behavior of a distributed system. This model is correct regarding the given requirements. The main advantage to similar work is that additionally required messages are identified and added to the system, while avoiding the addition of unnecessary messages. Furthermore, this thesis examines the application of the distributed synthesis approach to generate communication software for mechatronic systems. To this end, the approach was extended to support real-time requirements.

Developers usually cannot understand the full behavior of the system due to its complexity. Therefore, previous works have introduced scenario-based design approaches. Using these approaches, developers can concentrate on isolated situations in which only a limited number of requirements are relevant. The distributed synthesis approach described in this thesis operates on a scenario-based specification of requirements using Modal Sequence Diagrams. Furthermore, the thesis presents concepts for integrating the approach into the MechatronicUML development method. This method supports further development of the synthesized model to the final software.

Danksagung

An dieser Stelle möchte ich allen danken, die meine Arbeit an dieser Dissertation in irgendeiner Weise erleichtert haben.

Ich bedanke mich bei meinem Doktorvater Prof. Dr. Wilhelm Schäfer für die Aufnahme in die Fachgruppe Softwaretechnik und für die Betreuung dieser Arbeit. Besonders danke ich ihm für die Ermutigung, überhaupt eine Promotion anzustreben und dafür, dass ich immer genügend Freiraum hatte, an meinem selbst gewählten Forschungsthema zu arbeiten.

Ich danke Prof. Dr. Joel Greenyer dafür, dass er mit seiner eigenen Forschung die Grundlage für meine Arbeit gelegt hat. Außerdem danke ich ihm für viele hilfreiche fachliche und nicht-fachliche Gespräche in Paderborn, Mailand und via Skype.

Ich danke meinen ehemaligen Kollegen im Fachbereich Softwaretechnik für interessante Diskussionen während der Arbeit, aber auch in den Kaffeepausen: Anas Anis, Matthias Becker, Prof. Dr.-Ing. Steffen Becker, Christopher Brink, Nicola Danielzik, Stefan Dziwok, Markus Fockel, Jens Frieben, Dr. Christian Heinzemann, Jörg Holtmann, Thorsten Koch, Sebastian Lehrig, Renate Löffler, Ahmet Mehic, Dr. Matthias Meyer, Faruk Pasic, Marie Christin Platenius, Uwe Pohlmann, Dr. Claudia Priesterjahn, Dr. Jan Rieke, David Schmelter, Lars Stockmann, Christian Stritzke, Julian Suck, Oliver Sudmann, Dietrich Travkin, Dr. Markus von Detten, Benedikt Wohlers und Jinying Yu. Besonders danke ich den Teilnehmern der Diskussionsrunden zu MechatronicUML.

Ich danke Jutta Haupt, Astrid Canisius und Prof. Dr. Eckhard Steffen für ihre Hilfe in organisatorischen Fragen. Jürgen „Sammy“ Maniera danke ich für seine Hilfe im technischen Bereich.

Ich danke meinen Kooperationspartnern im ScenarioTools-Projekt für viele inspirierende Diskussionen und die gute Zusammenarbeit, besonders Prof. Dr. Joel Greenyer, Dr. Valerio Panzica La Manna, Jörg Holtmann und Grischa Liebel.

Schließlich danke ich meinen Eltern, meinen Großeltern und meinem Bruder für ihre Unterstützung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Software wird in zunehmendem Umfang in verschiedensten Arten von technischen Systemen eingesetzt, um sie sicherer, effizienter oder vielseitiger zu machen. Ihr Einsatz zur Kommunikation, Information und Unterhaltung ist den meisten Menschen vertraut. Immer wichtiger für unser Leben wird jedoch Software in Maschinen, die bisher rein mechanisch funktionierten. Viele industrielle Vorgänge, beispielsweise in der Produktion und der Logistik, laufen heute durch Softwaresteuerung automatisiert ab. Menschen vertrauen Software schon lange in Luft- und Raumfahrt ihr Leben an und überlassen mittlerweile auch im Automobil immer wichtigere Funktionen ihrer Kontrolle. Weitere Beispiele finden sich etwa in der Medizintechnik oder im Militär.

Verteilte mechatronische Systeme Ursprünglich war die Entwicklung technischer Systeme ausschließlich Gegenstand des Maschinenbaus, später auch von Elektrotechnik und Regelungstechnik. Je mehr jedoch Software das Systemverhalten steuert, umso mehr wird auch die Softwaretechnik unverzichtbar. Die Kombination dieser vier Disziplinen bezeichnet man als *Mechatronik*, die durch sie entwickelten Systeme nennt man *mechatronische Systeme*.

Viele mechatronische Systeme können im Falle von Fehlfunktionen durch ihre mechanischen Anteile große Sachschäden verursachen oder sogar Menschen verletzen. Es handelt sich dann um *sicherheitskritische Systeme*, bei denen ein inkorrektes Systemverhalten nicht toleriert werden kann. Da das Systemverhalten maßgeblich durch die Software bestimmt wird, muss sichergestellt werden, dass diese korrekt funktioniert. Das heißt, dass sie in allen Situationen die geforderten *Anforderungen* erfüllen muss. Speziell mechatronische Systeme müssen dabei auch *Echtzeitanforderungen* einhalten. Diese fordern, dass das System nicht nur auf die richtige Weise agiert, sondern auch zum richtigen Zeitpunkt.

Da mechatronische Systeme in steigendem Maße eingesetzt werden, operieren sie immer seltener isoliert von anderen Systemen. Stattdessen sind sie meist Teil eines komplexeren *verteilten Systems* und müssen sich daher mit anderen Subsystemen koordinieren oder sogar mit ihnen kooperieren. Fortschrittliche Funktionen wie beispielsweise Fahrerassistenzsysteme sind oft erst durch das Zusammenspiel mehrerer Subsysteme realisierbar. Zur Kooperation oder Koordination der Subsysteme müssen diese Informationen austauschen, was Kommunikation erfordert.

Das Systemverhalten kann maßgeblich von den übermittelten Informationen abhängen. Daher kann das Verhalten der einzelnen Subsysteme nicht mehr isoliert betrachtet werden. Implizite Abhängigkeiten und Wechselwirkungen führen zu einem komplexen Gesamtverhalten, das die Entwickler des Systems nur schwer verstehen oder analysieren

können. Dies kann leicht dazu führen, dass sie die Implementierung fehlerhaft durchführen, was insbesondere bei sicherheitskritischen Systemen nicht toleriert werden kann.

Da sich die Anforderungen auf das Gesamtverhalten des Systems beziehen können, müssen die Entwickler trotz dieser Komplexität in die Lage versetzt werden, die einzelnen Subsysteme so zu entwickeln, dass das Verhalten des resultierenden verteilten Systems diesen globalen Anforderungen genügt. Dabei sollten menschliche Fehlentscheidungen so weit wie möglich ausgeschlossen werden. Daher werden Methoden und Werkzeuge benötigt, die den Übergang von den globalen Anforderungen zur korrekten verteilten Software durch Automatisierung erleichtern.

Szenariobasierte Synthese von Verhaltensmodellen Um Anforderungen automatisiert verarbeiten und analysieren zu können, müssen diese zunächst durch Entwickler formal spezifiziert werden. Dazu müssen die Entwickler die Anforderungen jedoch verstehen können, was durch die bereits angesprochene Komplexität des Systemverhaltens erschwert wird. Aus diesem Grund wurden *szenariobasierte Spezifikationstechniken* entwickelt, bei denen Entwickler die Anforderungen aufgeteilt in mehrere *Szenarien* modellieren. Jedes Szenario beschreibt für mehrere Subsysteme in einer konkreten Situation nur das Verhalten, das in dieser Situation relevant ist. Die Entwickler werden somit jeweils nur mit einem Teil der Komplexität des Gesamtsystems konfrontiert.

Um Szenarien intuitiv und zugleich formal beschreiben zu können, wurden formale Varianten der *Message Sequence Charts (MSCs)* [Int11] entwickelt. Mit MSCs kann der Nachrichtenaustausch zwischen Subsystemen graphisch dargestellt werden. MSCs wurden in Form der bekannten Sequenzdiagramme in die *Uniform Modeling Language (UML)* [Obj11] übernommen, deren Semantik jedoch nur informell beschrieben ist. Eine verbreitete Formalisierung der MSCs sind die *Live Sequence Charts (LSCs)* [DH01, HM03], die in Form der *Modal Sequence Diagrams (MSDs)* [HM08] auch als formale Variante der UML-Sequenzdiagramme definiert wurden. Diese Arbeit befasst sich mit Anforderungsspezifikationen auf Grundlage von MSDs.

Um alle Anforderungen einzuhalten, muss die Implementierung des Systems alle Szenarien und deren oft implizite Abhängigkeiten zueinander berücksichtigen. Diese Abhängigkeiten können bei einer manuellen Umsetzung durch die Entwickler leicht übersehen werden, was zu einer fehlerhaften Software führt, die die Anforderungen verletzt. Um solche Fehler zu vermeiden, sollte der Vorgang der Implementierung möglichst weitgehend automatisiert werden.

Durch ein als *Synthese* bezeichnetes Verfahren kann zu einer gegebenen formalen Spezifikation automatisiert ein Implementierungsmodell erzeugt werden, das diese einhält. Während die Spezifikation im Allgemeinen mehrere Alternativen für ein korrektes Systemverhalten vorsieht, definiert dieses Implementierungsmodell nur eine davon. Es kann als Grundlage für eine manuelle Weiterentwicklung und schließlich zur automatischen Codeerzeugung verwendet werden.

Im Falle eines verteilten Systems muss das Implementierungsmodell das Verhalten jedes einzelnen Subsystems definieren und enthält daher ein Verhaltensmodell für jedes von diesen. Dabei müssen diese Verhaltensmodelle sicherstellen, dass die Subsysteme zusammengenommen die Spezifikation einhalten. Insbesondere muss jedes dieser Ver-

haltensmodelle in der Lage sein, Situationen, in denen es sich unterschiedlich verhalten muss, voneinander zu unterscheiden. Abhängig von der gegebenen Spezifikation kann dazu zusätzliche Kommunikation zwischen den Subsystemen erforderlich sein, welche in der Spezifikation selbst nicht explizit definiert ist. Diese zusätzliche Kommunikation muss bei der Synthese erzeugt werden.

Einige bestehende Verfahren zur Synthese verteilter Systeme (z. B. von Harel und Kugler [HK02]) sind bereits in der Lage, zusätzlich benötigtes Kommunikationsverhalten zu erzeugen. Diese Verfahren berücksichtigen jedoch nicht, ob im konkreten Fall überhaupt Kommunikation erforderlich ist. Unnötige Kommunikation verringert jedoch die Systemperformanz und kann schnell das verwendete Netzwerk überlasten. Zudem kann sie zur Verletzung von Echtzeitanforderungen führen. Andere existierende Syntheseverfahren (z. B. Bontemps et al. [BHS05] oder Harel et al. [HKP05]) erzeugen keinerlei Kommunikationsverhalten, das nicht bereits von der Spezifikation vorgegeben ist. Dies schränkt die Anwendbarkeit dieser Verfahren auf solche Spezifikationen ein, bei denen die Entwickler jede erforderliche Kommunikation bereits explizit definiert haben. Je größer jedoch die Spezifikation wird, desto wahrscheinlicher wird es, dass die Entwickler einzelne Fälle übersehen. Weiterhin berücksichtigt keines der für verteilte Systeme verwendbaren Syntheseverfahren Echtzeitanforderungen.

Um automatisiert praxistaugliche Verhaltensmodelle für mechatronische Systeme aus szenariobasierten Anforderungen zu erzeugen, wird daher ein neuartiger Syntheseansatz benötigt. Dieser soll zusätzliches Kommunikationsverhalten möglichst nur bei Bedarf erzeugen, also wenn andernfalls kein spezifikationskonformes Verhalten möglich ist. Einschränkungen der möglichen Kommunikation durch die Architektur müssen dabei berücksichtigt werden. Der Ansatz muss weiterhin in der Lage sein, sicherzustellen, dass das erzeugte System auch Echtzeitanforderungen einhält.

1.1 Anwendungsbeispiel

Zur Veranschaulichung des in dieser Arbeit entwickelten Ansatzes wird ein durchgängiges Beispiel für ein verteiltes mechatronisches System verwendet. Es handelt sich um eine Produktionszelle, die aus Rohlingen in einem oder mehreren Verarbeitungsschritten Metallplatten erzeugt. Dieses Anwendungsbeispiel basiert auf einer von Lewerentz und Lindner [LL95] beschriebenen Fallstudie. Für eine Variante dieser Fallstudie wurde bereits eine szenariobasierte MSD-Spezifikation beschrieben [Gre11]. Das Beispiel in dieser Arbeit orientiert sich jedoch nur grob an diesen Vorarbeiten.

Abbildung 1.1 zeigt eine einfache Variante der Produktionszelle des Beispiels. Auf einem Zuführförderband kommen in unregelmäßigen Abständen Metallrohlinge an. Der vorderste Rohling auf dem Band wird von einem Sensor als intakt oder defekt erkannt und an einen Roboterarm gemeldet **(1)**. Dieser nimmt den Rohling vom Band und sortiert ihn aus, falls er defekt ist **(2a)**. Andernfalls bewegt er ihn zu einer Presse **(2b)**. Diese presst den Rohling und wandelt ihn dadurch in eine fertige Metallplatte um **(3)**. Nach jedem Pressvorgang bewegt der Roboterarm die fertiggestellte Metallplatte auf ein Ablageförderband **(4)**. Schließlich bewegt sich der Arm zurück zum Zuführförderband und wartet auf einen neuen Rohling, um mit ihm ebenfalls wie beschrieben zu verfahren.

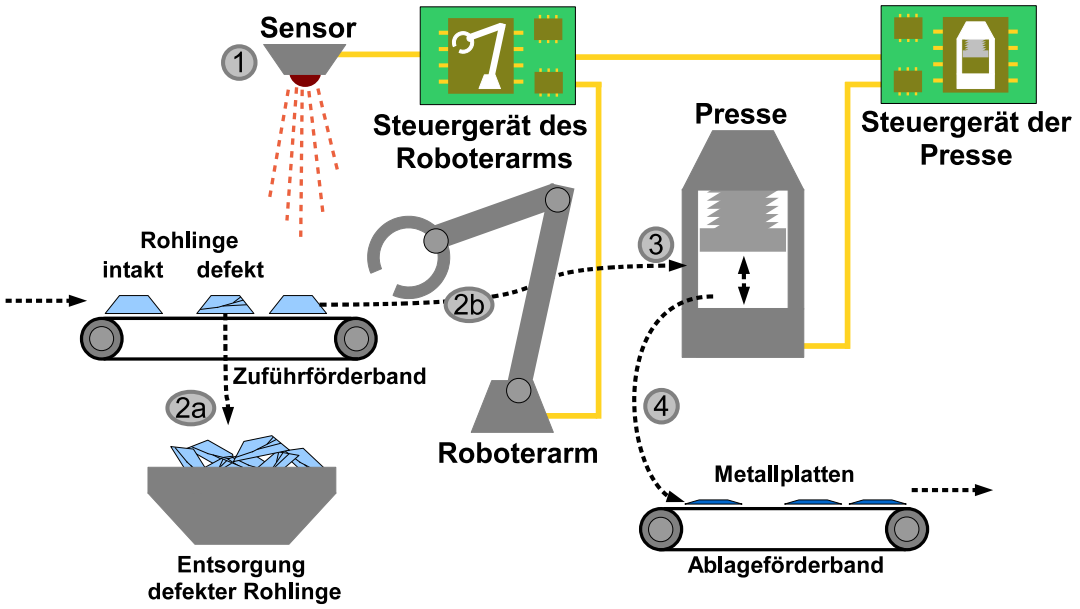


Abbildung 1.1: Skizze der Produktionszelle als Anwendungsbeispiel

Der Roboterarm und die Presse handeln nicht selbsttätig, sondern erhalten Befehle durch eigene Steuergeräte, mit denen zusammen sie jeweils autonome Subsysteme bilden. Diese Steuergeräte sind miteinander über einen Bus verbunden; an das Steuergerät des Arms ist zusätzlich der Sensor angeschlossen. Um die beschriebene Funktionalität umzusetzen, ist neben den Steuerbefehlen an Arm und Presse Kommunikation zwischen den beiden Steuergeräten erforderlich, um die Bewegungen des Arms und den Pressvorgang abzustimmen. Weiterhin meldet der Sensor des Zuführförderbands ankommende Rohlinge und deren Zustand (defekt/intakt) an das Steuergerät des Arms.

In Abschnitt 3.1 wird eine szenariobasierte Spezifikation der Anforderungen an diese Kommunikation mittels MSDs vorgestellt. Aufgabe des Syntheseverfahrens ist es, für das Kommunikationsverhalten beider Steuergeräte je ein Verhaltensmodell zu erzeugen, sodass das Gesamtsystem die in der Spezifikation festgelegten Anforderungen erfüllt.

In dieser Arbeit werden verschiedene Varianten des Produktionszellen-Beispiels verwendet, um verschiedene Aspekte des Verfahrens zu erläutern. Gegenüber der hier beschriebenen Basisvariante wird dabei die Produktionszelle um weitere Subsysteme ergänzt und die Modellierung der Umgebung wird erweitert.

1.2 Problemstellung

Um konsistente – also widerspruchsfreie – Anforderungen zu garantieren, beschreibt die Dissertation von Greenyer [Gre11] einen Ansatz zur Formalisierung von Anforderungen an mechatronische Systeme und zu deren automatischer Untersuchung auf Widersprüche. Die Formalisierung erfolgt dabei, wie in der vorliegenden Arbeit, durch szenariobasierte Modellierung der Anforderungen mittels der bereits erwähnten MSDs.

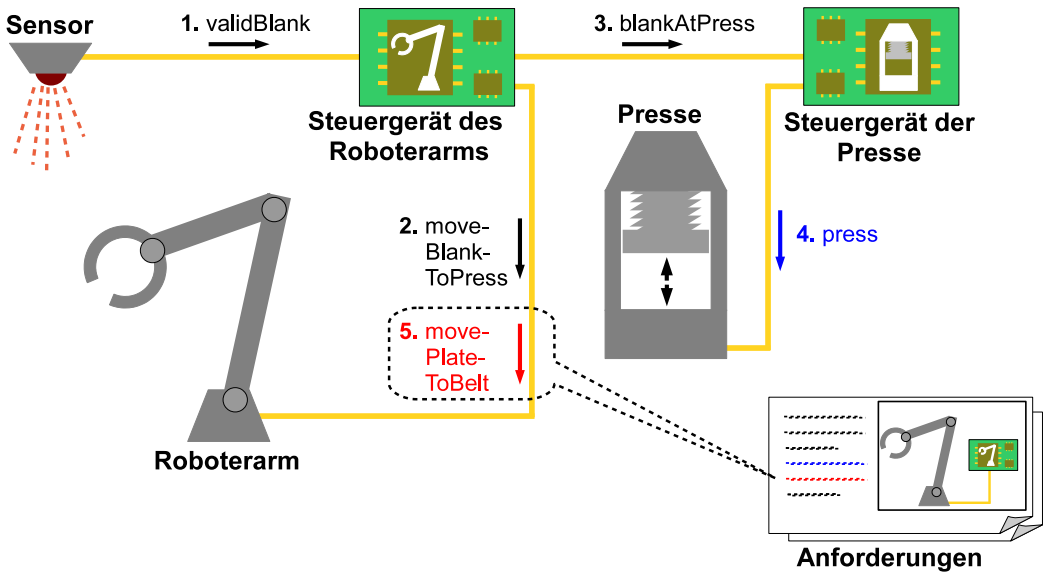


Abbildung 1.2: Beispiel für eine Situation, in der ein Steuergerät zusätzliche Informationen benötigt

Mit diesen können neben Anforderungen an das zu implementierende System auch Annahmen an seine Umgebung formalisiert werden, die zur Erfüllung der Anforderungen erforderlich sind.

Selbst eine formal definierte und widerspruchsfreie Spezifikation durch MSDs ist jedoch nicht zwangsläufig auch als verteiltes System realisierbar. Dies liegt daran, dass die Entwickler mittels MSDs zunächst aus Sicht eines externen „allsehenden“ Beobachters globale Anforderungen an die Kommunikation zwischen den Subsystemen definieren, während die einzelnen Subsysteme nur die selbst gesendeten oder empfangenen Nachrichten beobachten können. Kann ein Subsystem eine Nachricht jedoch nicht beobachten, dann ist es problematisch, wenn die Spezifikation von diesem Subsystem eine Reaktion auf die Nachricht verlangt. Beispielsweise beinhalten die Anforderungen an die in Abbildung 1.1 dargestellte Produktionszelle, dass der Roboterarm irgendwann eine Metallplatte zum Ablageförderband bewegen muss. Eine weitere Anforderung ist allerdings, dass dies erst dann passieren darf, wenn der Pressvorgang bereits durchgeführt wurde.

Insgesamt ergibt sich dadurch die in Abbildung 1.2 dargestellte Situation. Hier sind an den Netzwerkverbindungen zwischen den Subsystemen die zwischen diesen ausgetauschten Nachrichten eingezeichnet. Die Pfeile geben die Kommunikationsrichtung an.

Die problematische Situation ergibt sich nach Austausch der mit **1.** bis **4.** markierten Nachrichten. Die Anforderungen sehen nun vor, dass als nächstes die mit **5.** markierte Nachricht vom Steuergerät des Arms an ebendiesen gesendet wird. Diese würde die Bewegung einer Metallplatte von der Presse zum Ablageförderband einleiten. Das Steuergerät des Arms muss also auf die mit **4.** markierte Nachricht vom Steuergerät der Presse an die Presse selbst reagieren. Von dieser Nachricht hat es jedoch keine Kenntnis, da es weder Sender noch Empfänger ist.

Insgesamt impliziert die Spezifikation also, dass das Steuergerät des Arms irgendwann eine Nachricht erhalten muss, die es über die Durchführung des Pressvorgangs informiert. Dies kann eine bereits in der Spezifikation definierte Nachricht sein, wenn das Steuergerät des Arms aus ihrem Empfang schließen kann, dass der Pressvorgang durchgeführt wurde.

Gibt es keine solche Nachricht in der Spezifikation, dann muss eine entsprechende zusätzliche Nachricht im System eingeführt werden. Diese kann hier nur vom Steuergerät der Presse kommen, das über den Austausch der **4.** Nachricht informiert ist, da es diese selbst sendet. Wäre keine Kommunikation zwischen Arm-Steuergerät und Presse-Steuergerät möglich (also auch nicht indirekt über andere Subsysteme), so wäre die Spezifikation nicht verteilt realisierbar.

Bezieht man Echtzeitanforderungen in die Spezifikation mit ein und berücksichtigt man den Umstand, dass Kommunikation Zeit benötigt, dann kann eine Spezifikation in dem beschriebenen Fall auch dadurch unrealisierbar sein, dass die zusätzlich erforderliche Kommunikation zu lange dauert. Dies kann zur Verletzung der Echtzeitanforderungen führen. Beispielsweise könnte das Steuergerät des Arms erst mit deutlicher Verzögerung darüber informiert werden, dass der Pressvorgang durchgeführt wurde. Dadurch könnte es dann zu spät sein, sowohl eine Metallplatte zum Ablageförderband zu bewegen, als auch einen neuen Rohling vom Zuführförderband anzunehmen, bevor dieser am Ende des Bandes zu Boden fällt.

Sofern eine Spezifikation implementierbar ist, so ist der nächste Schritt das Erstellen eines Design-Modells für jedes zu implementierende Subsystem, welches das Kommunikationsverhalten des Subsystems beschreibt. Da die Modellierung der Design-Modelle bislang manuell erfolgt, ist es Aufgabe der Entwickler, zu erkennen, wann zusätzliches Kommunikationsverhalten – wie es im Beispiel vom Steuergerät des Arms benötigt wird – erforderlich ist. Je komplexer die Spezifikation ist, umso schwieriger ist es für die Entwickler, dies manuell festzustellen. Erschwerend kommt hinzu, dass alle Design-Modelle zusammengenommen konsistent dieselbe mögliche Implementierung der Spezifikation beschreiben müssen.

Wie bereits erwähnt wurde, existieren bereits Verfahren zur automatischen Erzeugung verteilter Design-Modelle. Diese generieren jedoch entweder auch dann zusätzliches Kommunikationsverhalten, wenn dieses nicht benötigt wird, oder sie generieren überhaupt keines. Ist für ein Verfahren letzteres der Fall, dann setzt dieses voraus, dass der Entwickler sämtliche zusätzlich erforderliche Kommunikation zuvor manuell in der Spezifikation ergänzt. Dies ist jedoch bei größeren Spezifikationen durch die hohe Komplexität problematisch.

Zudem unterstützen bisherige Ansätze zur verteilten Synthese keine Echtzeitanforderungen und keine asynchrone Kommunikation. Für eine detailliertere Behandlung verwandter Arbeiten wird auf Abschnitt 7 verwiesen.

Als offenes Problem kann hier also identifiziert werden, *dass der manuelle Übergang von den globalen Anforderungen mechatronischer Systeme zu einer verteilten Implementierung fehleranfällig ist, es bislang aber keinen praktikablen Ansatz gibt, der diesen Schritt automatisiert* und damit diese potentielle Fehlerquelle beseitigt.

1.3 Lösungsansatz und Beitrag der Arbeit

Diese Arbeit stellt einen Ansatz vor, der das im vorherigen Abschnitt beschriebene Problem löst. Der Hauptbeitrag ist ein Syntheseverfahren zur Automatisierung des Übergangs von szenariobasierten Spezifikationen zu verteilten Implementierungsmodellen. Der Ansatz erzeugt zu einer gegebenen Spezifikation ein verteiltes Implementierungsmodell, welches diese Spezifikation erfüllt.

Für die Anwendung des Verfahrens muss die Spezifikation dazu szenariobasiert mittels MSDs definiert werden. Das erzeugte Implementierungsmodell besteht aus einem Automaten für jedes zu implementierende Subsystem, der dessen Kommunikationsverhalten definiert. Er legt also fest, welche Nachrichten das Subsystem nach Empfang welcher Nachrichtensequenzen sendet. Das Verfahren identifiziert dabei automatisch zusätzlich benötigtes Kommunikationsverhalten und ergänzt dieses. Dabei berücksichtigt es architekturbedingte Einschränkungen der möglichen Kommunikationsverbindungen.

Um das Syntheseverfahren für mechatronische Systeme verwendbar zu machen, wurde es um die Unterstützung von Echtzeitverhalten erweitert. Dazu gehört die Berücksichtigung von Echtzeitanforderungen und -annahmen der Spezifikation, d. h. das erzeugte Implementierungsmodell hält diese Anforderungen ein, wenn die Annahmen gelten. Weiterhin werden architekturspezifische Annahmen über die Übertragungszeit von Nachrichten unterstützt. Zudem wurde das Verfahren um die Behandlung asynchroner Kommunikation erweitert, da diese in realen mechatronischen Systemen verbreitet ist.

Das Ergebnis des Syntheseverfahrens ist ein Implementierungsmodell des diskreten Kommunikationsverhaltens. Dieses kann im Prinzip, nach Ergänzung plattformspezifischer Informationen, durch Codegenerierung direkt in die fertige Software übersetzt werden. Typischerweise ist der Systementwurf mit der Erzeugung dieses Modells jedoch noch nicht abgeschlossen. Beispielsweise unterstützt der Syntheseansatz bestimmte Verhaltensbestandteile, wie Parameterwerte und Variablenzugriffe nur eingeschränkt und kontinuierliches Verhalten überhaupt nicht, auch weil diese Aspekte nur bedingt mittels MSDs modelliert werden können. Zudem muss zur Vervollständigung des Systemmodells oft zusätzliches internes Verhalten der Subsysteme ergänzt werden.

Zur praktischen Anwendung des vorgestellten Ansatzes ist es daher erforderlich, dass das erzeugte Implementierungsmodell durch die Entwickler des Systems erweitert werden kann. Dazu müssen das Syntheseverfahren und seine Ein- und Ausgaben in eine existierende Entwicklungsmethode eingebettet werden, die auch eine entsprechende Werkzeugunterstützung bietet. Diese Arbeit beschreibt daher die Einbettung des vorgestellten Verfahrens in die bestehende Entwicklungsmethode MechatronicUML.

Insgesamt kann das Syntheseverfahren damit nach dem in Abbildung 1.3 abstrakt dargestellten *Entwicklungsprozess* zur Entwicklung der Software mechatronischer Systeme verwendet werden. Zunächst (Schritt **1**) müssen die Anforderungen manuell mittels MSDs formalisiert werden. Dabei können möglicherweise bestehende Modelle als Grundlage verwendet werden. Wird beispielsweise nach der bekannten VDI-Richtlinie 2206 [VDI04] für den Entwurf mechatronischer Systeme entwickelt, so können aus der Spezifikation einer *Prinziplösung* [Fra06] – einem interdisziplinären Modell für den konzeptionellen Entwurf eines Systems – die für die Kommunikationssoftware relevanten Anforderungen abgeleitet werden [Gre11].

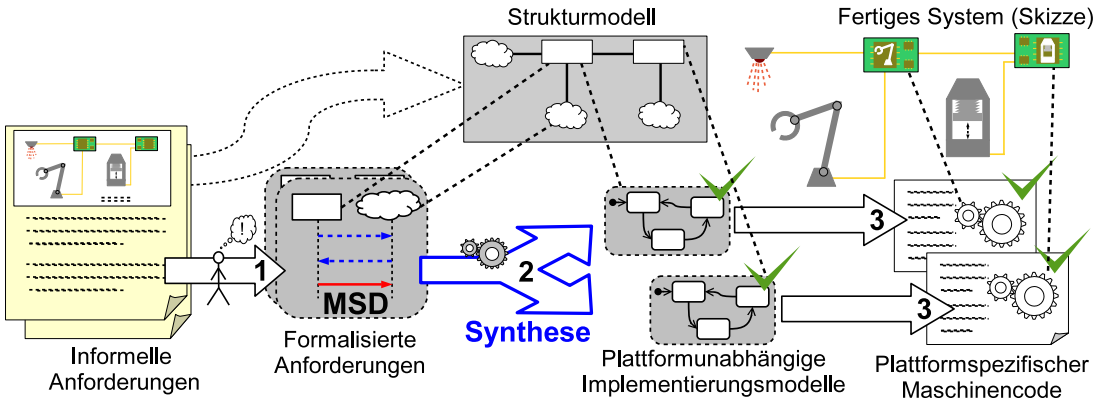


Abbildung 1.3: Überblick über den Entwicklungsprozess unter Berücksichtigung des verteilten Syntheseverfahrens

MSDs beziehen sich auf ein grobes Strukturmodell, das zwischen den zu entwickelnden Subsystemen (Rechtecke) und den Bestandteilen der Umgebung (wolkenförmige Symbole) unterscheidet. Das als Hauptbeitrag dieser Arbeit entwickelte Syntheseverfahren (Schritt **2**) erzeugt automatisch plattformunabhängige Implementierungsmodelle für diese Subsysteme, welche die durch die MSDs definierten Anforderungen einhalten. Die Implementierungsmodelle können anschließend im Rahmen der MechatronicUML-Entwicklungsmethode manuell verfeinert und um plattformspezifische Informationen erweitert werden, um schließlich durch Codegenerierung eine ausführbare Software für jedes der zu entwickelnden Subsysteme zu erhalten (Schritt **3**).

Der hier vorgestellte Prozess ist stark vereinfacht und bezieht sich nur auf die Entwicklung der diskreten Anteile der Kommunikationssoftware. In der Realität wird die Entwicklung mechatronischer Systeme insbesondere durch wechselseitige Beeinflussung der an der Entwicklung beteiligten Disziplinen sehr komplex [RDS+12]. Für eine ausführliche Betrachtung der Methoden und Prozesse, die für eine disziplinübergreifende Entwicklung mechatronischer Systeme erforderlich sind, wird auf Gausemeier et al. [GRS14, GFDK09] verwiesen. Ein disziplinübergreifender Entwicklungsprozess mit Fokus auf die Softwaretechnik findet sich in Heinzemann et al. [HSST13].

Zusammengefasst sind die drei Hauptbeiträge dieser Arbeit:

- Entwicklung und Evaluation eines praktikablen Verfahrens zur Erzeugung verteilter Automatenmodelle auf Basis von MSD-Spezifikationen, das zusätzlich benötigte Kommunikation automatisch ergänzt.
- Erweiterung des Syntheseverfahrens um Unterstützung für Echtzeitsysteme und asynchrone Kommunikation.
- Integration des Syntheseverfahrens in MechatronicUML.

Jeder dieser Punkte umfasst die Erarbeitung der Konzepte und eine prototypische Implementierung. Letztere ist einerseits in die Werkzeugumgebung ScenarioTools

für szenariobasierte Anforderungsspezifikation mittels MSDs integriert. Andererseits erzeugt sie aber Modelle, die in der MechatronicUML Tool Suite, der Werkzeugumgebung für MechatronicUML weiterentwickelt werden können. Weiterhin wurden die durch die Prototypen umgesetzten Verfahren anhand einfacher Fallstudien auf Basis des Produktionszellen-Beispiels (vgl. Abschnitt 1.1) evaluiert.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert. Kapitel 2 beschreibt die Grundlagen zu MSD-Spezifikationen und den durch die Synthese zu erzeugenden Controllern. Kapitel 3 stellt das im Rahmen der Arbeit entwickelte Syntheseverfahren in der Basisvariante vor. Darauf aufbauend beschreibt Kapitel 4 die Erweiterungen um Echtzeit und asynchrone Kommunikation. Die Einbettung des so erweiterten Syntheseverfahrens in MechatronicUML wird in Kapitel 5 behandelt. Kapitel 6 beschreibt die Implementierung der in den vorangehenden drei Kapiteln vorgestellten Konzepte und deren Evaluierung anhand von Fallstudien. Verwandte Ansätze werden in Kapitel 7 behandelt, während Kapitel 8 schließlich die Arbeit zusammenfasst und einen Ausblick auf mögliche weiterführende Arbeiten gibt.

Kapitel 2

Grundlagen

Dieses Kapitel erläutert die für das Verständnis der nachfolgenden Kapitel relevanten Grundlagen. Diese werden insbesondere für Kapitel 3 und 4 benötigt, in denen das Syntheseverfahren bzw. dessen zeitbehaftete Erweiterung anhand beispielhafter Spezifikationen erläutert werden.

Das Kapitel ist wie folgt unterteilt: Abschnitt 2.1 erläutert, wie Anforderungen und Annahmen auf Grundlage von Modal Sequence Diagrams (MSDs) modelliert werden können und diskutiert die zugrundeliegenden Konzepte. Abschnitt 2.2 führt Controllersysteme ein, die das Kommunikationsverhalten eines Systems zustandsbasiert beschreiben. Controllersysteme werden durch den in dieser Arbeit beschriebenen Syntheseansatz als Implementierungen von MSD-Spezifikationen erzeugt.

2.1 Spezifikationen auf Basis von Modal Sequence Diagrams

Dieser Abschnitt erläutert die Grundlagen der formalen Spezifikation von Anforderungen an ein zu entwickelndes mechatronisches System und Annahmen an dessen Umgebung mittels *Modal Sequence Diagrams (MSDs)* [HM08]. MSDs sind eine Variante der *Live Sequence Charts (LSCs)* [DH98, DH01, HM03], die auf den Sequenzdiagrammen aus der UML basieren und diese durch ein UML-Profil um zusätzliche Arten von Modellelementen erweitern.

Bei MSDs kann zwischen *existentiellen MSDs* und *universellen MSDs* unterschieden werden. Erstere müssen für mindestens eine Ausführung des Systems gelten, letztere für alle möglichen Ausführungen. Diese Arbeit betrachtet ausschließlich universelle MSDs, da der vorgestellte Syntheseansatz eine Implementierung erzeugen soll, welche die Anforderungen für alle möglichen Ausführungen einhält. Hierfür sind existentielle MSDs nicht erforderlich.

MSDs wurden um die Möglichkeit erweitert, neben Anforderungen an das Systemverhalten auch Annahmen an das Verhalten der zukünftigen Einsatzumgebung des Systems zu definieren [Gre11, BGP13]. Anforderungen werden dabei in *Requirement MSDs* definiert, Annahmen in *Assumption MSDs*.

Da MSDs Anforderungen (bzw. Annahmen) jeweils bezogen auf konkrete Subsysteme (bzw. Bestandteile der Umgebung) definieren, müssen diese Strukturelemente zunächst definiert werden. In dieser Arbeit wird zur Definition der Systemstruktur eine Teilmenge der UML [Obj11] mit einigen Anpassungen verwendet. Die Gesamtheit der MSDs für ein System wird zusammen mit der Strukturdefinition als *MSD-Spezifikation* bezeichnet.

Die Strukturdefinition wird in Abschnitt 2.1.1 beschrieben. Auf dieser Basis werden dann zunächst in Abschnitt 2.1.2 die grundlegende Syntax und Semantik der MSDs er-

läutert. Die Diskussion beschränkt sich dabei zunächst auf Requirement MSDs, während die Definition von Umgebungsannahmen mittels Assumption MSDs in Abschnitt 2.1.3 thematisiert wird. Um bestimmte Verhaltensaspekte einfacher und andere überhaupt erst beschreiben zu können, werden in Abschnitt 2.1.4 zusätzliche spezielle Modellelemente für MSDs eingeführt. Abschnitt 2.1.5 behandelt zeitbehaftete Modellelemente, mit denen MSDs Echtzeitanforderungen beschreiben können. Abschließend werden in Abschnitt 2.1.6 die Kriterien definiert, unter denen ein gegebenes System eine korrekte Implementierung einer MSD-Spezifikation ist.

2.1.1 Strukturdefinition: UML

Die Struktur des Systems, auf das sich eine MSD-Spezifikation bezieht, wird in dieser Arbeit mittels der UML-Diagramme [Obj11] Klassendiagramm und Kompositionsstrukturdiagramm definiert. Dabei werden Klassendiagramme zur Definition der möglichen Typen von Subsystemen verwendet, während Kompositionsstrukturdiagramme die konkreten im System vorhandenen Instanzen und die Kommunikationsverbindungen zwischen diesen definieren. Weder System noch Umgebung können Nachrichten senden, für die in der Strukturdefinition keine geeignete Kommunikationsverbindung definiert ist.

Abbildung 2.1 zeigt ein Beispiel für eine Strukturdefinition. Unten im Bild zeigt eine Skizze informell einen Ausschnitt der Struktur des in Abschnitt 1.1 eingeführten Beispiels. In der hier gezeigten Variante enthält das System jedoch statt einem zwei Roboterarme und für jeden von diesen ein eigenes Steuergerät. Zudem verfügt eines der Steuergeräte über ein Benutzerinterface.

Die Steuergeräte im Beispiel sind über ein Netzwerk einerseits mit dem jeweils von ihnen gesteuerten Roboterarm, andererseits mit einem Sensor verbunden. Das Steuergerät mit Benutzerinterface hat auch zu diesem eine Kommunikationsverbindung. An den Verbindungen zwischen den Subsystemen sind jeweils die ausgetauschten Nachrichten und die Richtung der jeweiligen Kommunikation eingezeichnet. Die übrigen Bildelemente werden nachfolgend schrittweise erläutert.

Zunächst wird erläutert, wie die Bestandteile des Systems auf Typebene durch Klassendiagramme definiert werden können. Anschließend wird beschrieben, wie die Struktur des Systems durch Kompositionsstrukturdiagramme definiert wird.

Definition der Typen von Systembestandteilen

Ein *Klassendiagramm* (Abbildung 2.1 oben) definiert die möglichen Typen von Subsystemen in Form von *Klassen*. Im Beispiel sind dies Sensor, UI, ArmController und Arm. Zudem definieren die *Operationen* der Klassen die Nachrichten, welche von Instanzen der jeweiligen Klasse empfangen werden können.

Im Beispiel kann Arm die Nachrichten `blankToPress`, `removeBlank` und `plateToBelt` empfangen, die jeweils einen speziellen Befehl an den Roboterarm repräsentieren. Während eine gegebene Nachricht also nur von Instanzen bestimmter Klassen empfangen werden kann, kann prinzipiell jede Instanz einer beliebigen Klasse sie senden. Im Einzelfall kann jedoch die Kommunikationsstruktur des Systems (siehe nächster Abschnitt) dies verhindern.

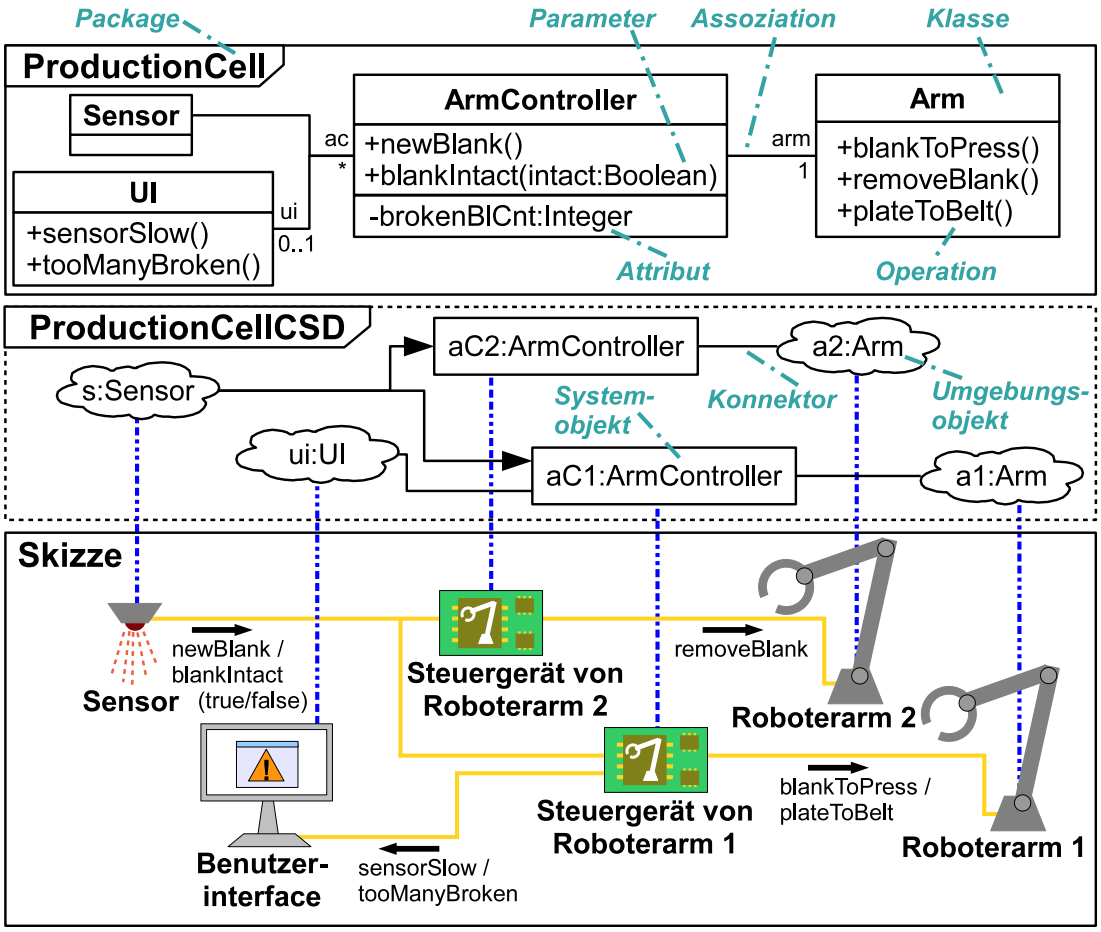


Abbildung 2.1: Beispiel für eine Strukturdefinition durch Klassen auf Typebene (oben) und Objekte auf Instanzebene (Mitte) für das unten als informelle Skizze gezeigte reale System

Operationen können einen oder mehrere *Parameter* haben, die jeweils durch einen Namen und einen Typ definiert sind. Nachrichten, die einer parametrisierten Operation entsprechen, müssen dann Werte des passenden Typs für jeden dieser Parameter definieren. Im Beispiel hat die Operation blankIntact der Klasse ArmController den Parameter `intact` des Typs Boolean. Eine konkrete, zur Laufzeit übertragene Nachricht `blankIntact` muss dann einen Wert von entweder `true` oder `false` für diesen Parameter definieren.

Neben Operationen können Klassen auch *Attribute* und *Assoziationen* definieren, auf welche die MSDs der Spezifikation Bezug nehmen können. Dies wird in Abschnitt 2.1.4 erläutert. Im Klassendiagramm in Abbildung 2.1 (oben) wird mittels der Assoziation arm beispielsweise modelliert, dass jedem ArmController genau ein Arm zugeordnet ist. Das Beispiel definiert ein Attribut namens brokenBICnt für die Klasse ArmController, dessen Typ Integer ist.

Systembestandteile und Kommunikationsstruktur

Die Bestandteile des Systems, auf welches sich eine MSD-Spezifikation bezieht, und von dessen Umgebung werden durch *Objekte* modelliert. Dies sind Instanzen der Klassen, die für die MSD-Spezifikation definiert wurden. Die im konkreten System vorhandenen Objekte werden durch ein *Kompositionsstrukturdiagramm* (CSD, für engl. *composite structure diagram*) definiert (siehe Abbildung 2.1 Mitte). Ein CSD legt fest, welche Objekte zu implementierende Subsysteme und welche Objekte Bestandteile der Umgebung repräsentieren. Weiterhin definiert es, welche von diesen Objekten miteinander kommunizieren können.

In einem Kompositionsstrukturdiagramm werden die Objekte, aus denen das System besteht, durch *Parts* repräsentiert. Diese sind durch die im Klassendiagramm definierten Klassen getypt. Das in Abbildung 2.1 gezeigte CSD legt beispielsweise fest, dass sich im System die Objekte aC1 und aC2 befinden, die jeweils Instanzen der Klasse ArmController sind. Im Rahmen dieser Arbeit repräsentieren Parts immer Objekte. Daher wird im Folgenden nicht zwischen Parts und den von ihnen repräsentierten Objekten unterschieden und stattdessen nur letzterer Begriff verwendet.

Weiterhin definiert das CSD, welche Objekte Subsysteme des zu implementierenden Systems und welche Objekte Bestandteile der Umgebung sind. Objekte, die direkt durch die zu entwickelnde Software kontrolliert werden können, werden im Diagramm rechteckig dargestellt. Diese Objekte werden auch als *Systemobjekte* bezeichnet. Die durch die Software nicht direkt kontrollierbaren Objekte werden als *Umgebungsobjekte* bezeichnet. Abweichend zur üblichen UML-Syntax werden diese Objekte im Diagramm wolkenförmig dargestellt [Gre11]. Im Beispiel in Abbildung 2.1 (Mitte) sind also die Objekte aC1 und aC2 Systemobjekte, während a1, a2, ui und s Umgebungsobjekte sind.

Die Verbindungslinien zwischen den Objekten im Diagramm werden als *Konnektoren* bezeichnet. Konnektoren repräsentieren Kommunikationsverbindungen, d. h. sie definieren, dass zwischen den durch sie verbundenen Objekten eine Kommunikation möglich ist. Dies ist sonst nicht der Fall. Nach dem CSD in Abbildung 2.1 kann also beispielsweise aC1 mit a1 kommunizieren, aber nicht a1 mit a2.

Im Unterschied zur UML werden hier auch gerichtete Konnektoren verwendet, die an einem Ende eine Pfeilspitze aufweisen. Diese definieren, dass Nachrichten nur in Pfeilrichtung gesendet werden können, d. h. das Objekt, auf das der Pfeil zeigt, muss der Empfänger sein. Im Beispiel kann s also Nachrichten an aC1 und aC2 senden, aber nicht umgekehrt.

Wenn in dieser Arbeit Objekte zur Laufzeit dargestellt werden, so werden diese innerhalb von *Objektdiagrammen* gezeigt. Für jedes in diesen enthaltene Objekt wird der Name des Objekts und die instanziierte Klasse in der Form `Objektname:Klassenname` angegeben, wie es in Abbildung 2.3 (Mitte) gezeigt wird, die im nachfolgenden Abschnitt näher beschrieben wird. Im Unterschied zu Kompositionsstrukturdiagrammen zeigen Objektdiagramme die Assoziationen zwischen Objekten, aber nicht die Kommunikationsverbindungen. Anders als Kompositionsstrukturdiagramme und Klassendiagramme werden Objektdiagramme nicht zur *Definition* der Struktur eingesetzt und sie sind nicht selbst Teil der MSD-Spezifikation.

2.1.2 Anforderungen: Modal Sequence Diagrams

Ein Modal Sequence Diagram (MSD) bezieht sich auf Nachrichten, die zwischen bestimmten Objekten ausgetauscht werden. Es definiert über diese Nachrichten eine Halbordnung, d. h. die Nachrichten dürfen nur in bestimmten Reihenfolgen auftreten. Tritt eine von der Halbordnung abweichende Reihenfolge auf, so ist dies eine Verletzung von *Safety-Anforderungen* an das System. Safety-Anforderungen definieren fehlerhaftes Systemverhalten, das nicht auftreten darf. Neben Safety-Anforderungen können MSDs auch *Liveness-Anforderungen* definieren. Diese definieren ein Systemverhalten, das irgendwann auftreten muss, da es zum korrekten Funktionieren des Systems erforderlich ist.

Im Folgenden werden zunächst Nachrichtenereignisse als Grundlage der MSD-Semantik eingeführt. Anschließend werden erst die Syntax und dann die Semantik der MSDs behandelt.

Kommunikationsverhalten zur Laufzeit: Nachrichtenereignisse

MSDs beziehen sich auf den Austausch von Nachrichten zwischen Objekten zur Laufzeit. Ein solcher Nachrichtenaustausch wird als *Nachrichtenereignis* (engl. *message event*), oder einfach als *Ereignis* (engl. *event*), bezeichnet. Eine Folge von Nachrichtenereignissen im System wird als ein *Ablauf* (engl. *run*) des Systems bezeichnet. Fasst man das Senden und das Empfangen einer Nachricht als ein einziges Ereignis auf, dann wird das als *synchrone Kommunikation* bezeichnet. Ist das Senden und das Empfangen jeweils ein eigenes Ereignis und können dazwischen weitere Ereignisse auftreten, dann ist das *asynchrone Kommunikation*.

Handelt es sich bei dem Objekt, das eine Nachricht sendet, um ein Systemobjekt, so handelt es sich um eine *Systemnachricht*, andernfalls um eine *Umgebungsnachricht*. Systemnachrichten werden auch als *kontrollierbare Nachrichten*, Umgebungsnachrichten als *unkontrollierbare Nachrichten* bezeichnet. Nachrichten, die das System sendet *oder* empfängt, werden als *beobachtbare Nachrichten* bezeichnet, die übrigen als *unbeobachtbare Nachrichten*.

Grundlegende Syntax von MSDs

Abbildung 2.2 zeigt das MSD ArmRemovesBrokenBlank, das Anforderungen an die Subsysteme des in Abbildung 2.1 gezeigten Systems definiert. Die linke obere Ecke des Diagramms beinhaltet den Namen des MSDs. Die einzelnen Elemente der Abbildung werden in diesem Abschnitt schrittweise zur Veranschaulichung der im Text beschriebenen Konzepte erläutert.

Die vertikalen, gestrichelten Linien mit entweder einem Rechteck oder einem wolkenförmigen Symbol am oberen Ende werden als *Lifelines* bezeichnet. Sie repräsentieren Objekte im System, die durch den Text im jeweiligen Symbol referenziert werden. Die hier gezeigte Variante von Lifelines ist die üblichste, die immer genau ein Objekt im System repräsentiert. Der Text hat dann die Form `Objektname:Klassenname` und identifiziert damit direkt das repräsentierte Objekt. Im Falle von Umgebungsobjekten – wie dem Sensor s im Beispiel – ist das Symbol wolkenförmig; bei Systemobjekten – wie dem ArmController aC1 – ist es ein Rechteck.

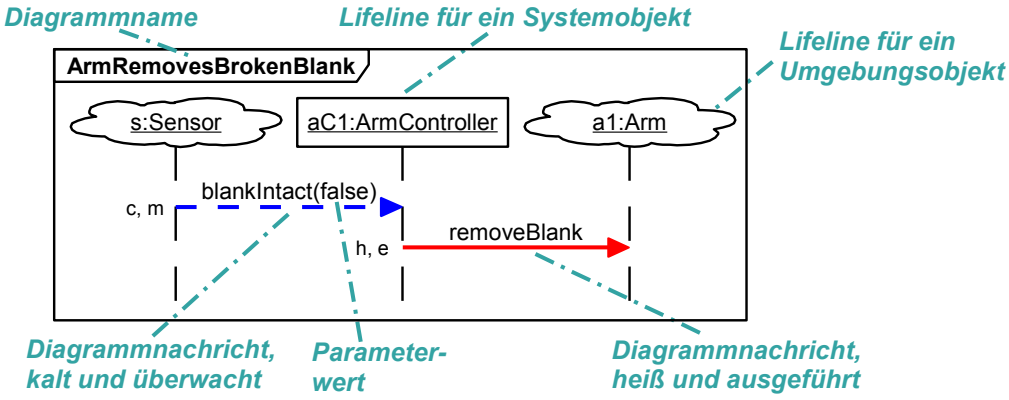


Abbildung 2.2: Beispiel für ein Requirement MSD

Die horizontalen Pfeile in den MSDs repräsentieren Nachrichten, die im System zwischen Objekten ausgetauscht werden. Die Pfeile werden als *Diagrammnachricht* oder abkürzend ebenfalls als *Nachricht* bezeichnet.

Der Pfeil jeder Nachricht geht von der Lifeline aus, die das sendende Objekt repräsentiert und zeigt auf die Lifeline des empfangenden Objekts. Er ist mit dem Namen der durch die jeweilige Nachricht referenzierten Operation beschriftet. Beispielsweise bezieht sich die oberste Nachricht des MSDs in Abbildung 2.2 auf die Operation blankIntact der Klasse ArmController (vgl. Abbildung 2.1). Hat die Operation *Parameter*, so muss in Klammern hinter dem Namen für jeden der Parameter entweder ein konkreter Wert oder ein Variablenname angegeben werden. Der Typ des Werts bzw. der Variable muss identisch zu dem des Parameters sein. Beispielsweise hat die Operation blankIntact der Klasse ArmController einen Parameter intact mit Typ Boolean. Daher muss für die entsprechende Nachricht im MSD ebenfalls ein Boolean-Wert angegeben werden. Dies geschieht im MSD ArmRemovesBrokenBlank durch Angabe des Parameterwerts `false`.

Diagrammnachrichten haben eine *Temperatur* und eine *Ausführungsart*. Temperatur und Ausführungsart der Nachrichten in einem MSD entscheiden darüber, ob die Anforderungen verletzt werden, wenn die im MSD definierten Nachrichten im realen System nicht oder nicht in derselben Reihenfolge auftreten. Diese Zusammenhänge werden hier nur informell skizziert und detaillierter im nachfolgenden Abschnitt erläutert.

Die Temperatur einer Nachricht ist entweder *heiß* oder *kalt*. Eine heiße Nachricht wird durch einen roten Pfeil dargestellt, eine kalte durch einen blauen Pfeil. Eine heiße Nachricht definiert eine Safety-Anforderung: wenn diese im MSD „an der Reihe“ ist, dann darf keine andere Nachricht des MSDs auftreten. Wenn statt einer kalten Nachricht eine andere auftritt, dann verletzt dies nicht die Anforderungen.

Die Ausführungsart ist entweder *ausgeführt* (engl. *executed*) oder *überwacht* (engl. *monitored*). Eine ausgeführte Nachricht wird durch einen durchgezogenen Pfeil dargestellt, während der Pfeil einer überwachten Nachricht gestrichelt ist. Eine ausgeführte Nachricht definiert eine Liveness-Anforderung: wenn diese im MSD „an der Reihe“ ist, dann muss sie irgendwann gesendet werden. Eine überwachte Nachricht andererseits muss nie gesendet werden.

Neben der Darstellung des Pfeils werden Temperatur und Ausführungsart zusätzlich textuell rechts oder links neben dem Pfeil dargestellt. Dabei steht h für heiß (engl. ***h**ot*), c für kalt (engl. ***c**old*), sowie m für überwacht (***m**onitored*) und e für ausgeführt (***e**xecuted*). Damit ergeben sich die möglichen Kombinationen „h, e“, „c, e“, „h, m“ und „c, m“. Beispielsweise ist im MSD ArmRemovesBrokenBlank die Nachricht `blankIntact` kalt und überwacht, während `removeBlank` heiß und ausgeführt ist.

Grundlegende Semantik von MSDs

Die Semantik von MSDs wird hier anhand der in Abbildung 2.3 gezeigten Beispielsituation im durch die MSD-Spezifikation beschriebenen Produktionssystem erläutert. Die Abbildung zeigt in der Mitte ein Objektdiagramm, welches die Systemstruktur zur Laufzeit darstellt. Diese Struktur entspricht der in Abbildung 2.1 definierten. Über und unter dem Objektdiagramm zeigt die Abbildung zwei MSDs und ihren aktuellen Ausführungszustand. Dieser wird nachfolgend zusammen mit den übrigen Elementen der Abbildung erläutert.

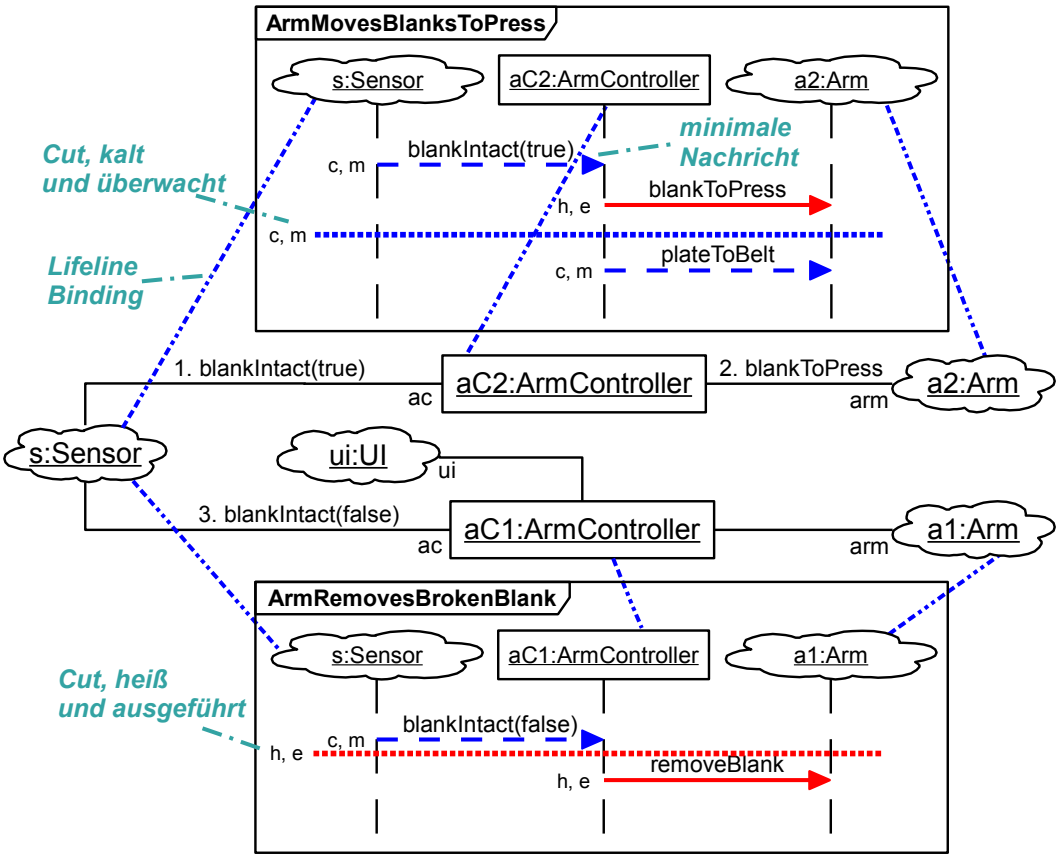


Abbildung 2.3: Beispiel für zwei aktive MSDs mit jeweiligem Cut und Lifeline Bindings (unten und oben) und das zugehörige System zur Laufzeit (Mitte)

Die Lifelines der in Abbildung 2.3 gezeigten MSDs beziehen sich auf die Objekte des Objektdiagramms zwischen diesen. Diese Beziehung der Lifelines eines MSDs zu je einem Objekt wird als *Lifeline Binding* bezeichnet. In der Abbildung sind die Lifeline Bindings als gestrichelte Linien eingezeichnet. Wie man am Sensor s sehen kann, können sich mehrere Lifelines gleichzeitig auf ein Objekt beziehen.

Diejenigen Diagrammnachrichten, über denen sich auf Sender- und Empfänger-Lifeline keine anderen Nachrichten befinden, werden als *minimale Nachrichten* bezeichnet. In dieser Arbeit wird davon ausgegangen, dass es in jedem MSD nur eine minimale Nachricht gibt. Minimale Nachrichten sind immer kalt und überwacht. Sie schränken selbst nicht das erlaubte Kommunikationsverhalten ein, sondern ihr Auftreten ist Bedingung dafür, dass der Rest des MSDs überhaupt relevant wird. In Abbildung 2.3 ist in beiden MSDs die Nachricht `blankIntact` minimal, jedoch jeweils mit anderem Parameter.

Beim Auftreten einer minimalen Nachricht wird eine neue Instanz des MSDs erzeugt, die als *aktives MSD* bezeichnet wird. Aktive MSDs haben keine eigene graphische Repräsentation. Sie werden vollständig durch das durch sie instanziierte MSD, eine Menge von Lifeline Bindings und einen aktuellen Ausführungszustand charakterisiert.

Der Ausführungszustand eines aktiven MSDs wird als *Cut* bezeichnet. Er kennzeichnet, welche Nachrichten des MSDs bereits im System gesendet wurden – unter Einhaltung einer durch das MSD definierten Reihenfolge (eine Halbordnung). Entsprechend dieser Reihenfolge kann eine Nachricht nach der minimalen Nachricht des MSDs nur vorkommen, wenn bereits alle Nachrichten übermittelt wurden, die auf den Lifelines von Sender und Empfänger dieser Nachricht oberhalb von ihr stehen. Abbildung 2.3 zeigt eine Situation in der Ausführung des Systems, in der die drei im Objektsystem eingezeichneten Nachrichtenereignisse `blankIntact(true)`, `blankToPress` und `blankIntact(false)` bereits in dieser Reihenfolge gesendet wurden. Dementsprechend befindet sich der Cut im MSD ArmMovesBlanksToPress vor `plateToBelt` und in ArmRemovesBrokenBlank befindet sich der Cut vor der Nachricht `removeBlank`. In der Abbildung ist der Cut in beiden MSDs jeweils als horizontale Linie eingezeichnet.

Der Cut definiert die der Reihenfolge des MSDs nach möglichen nächsten Nachrichten. Diese Nachrichten werden als *freigegeben* (engl. *enabled*) bezeichnet. Ist mindestens eine freigegebene Nachricht ausgeführt, dann wird der Cut als *ausgeführter Cut* bezeichnet, andernfalls als *überwachter Cut*. Ist mindestens eine der freigegebenen Nachrichten heiß, dann handelt es sich um einen *heißen Cut*, andernfalls um einen *kalten Cut*. In Abbildung 2.3 ist beispielsweise im MSD ArmMovesBlanksToPress als einzige Nachricht `plateToBelt` freigegeben, der Cut somit ebenfalls kalt und überwacht. In ArmRemovesBrokenBlank ist der Cut jedoch ausgeführt und heiß, da die Nachricht `removeBlank` mit denselben Eigenschaften freigegeben ist. In der Abbildung ist die Farbe der Linie des Cut entsprechend der Temperatur der Nachricht gewählt.

Tritt ein Ereignis auf, das einer freigegebenen Nachricht entspricht, so bewirkt dies, dass sich die Menge der freigegebenen Nachrichten im aktiven MSD ändert und der Cut sich entsprechend nach unten verschiebt. Ist der Cut dadurch auf allen Lifelines am Ende angelangt, d. h. zu allen Diagrammnachrichten sind entsprechende Ereignisse aufgetreten, dann terminiert das aktive MSD. Der Fall, dass statt einer freigegebenen eine nicht freigegebene Nachricht des MSDs auftritt, wird nachfolgend behandelt.

Violations Tritt ein Nachrichtenereignis im System auf, das einer nicht freigegebenen Diagrammnachricht in einem aktiven MSD entspricht, dann wird dies als *Violation* (Verletzung) des MSDs bezeichnet. Dies bedeutet, dass im System eine Nachrichtenabfolge aufgetreten ist, die der durch das MSD definierten Halbordnung über die Nachrichtenereignisse widerspricht. Abhängig von der Temperatur des aktuellen Cuts kann es sich bei einer Violation um gültiges Systemverhalten oder aber um eine Verletzung der Anforderungen handeln.

Ist das MSD bei Auftreten einer Violation in einem heißen Cut, so handelt es sich um eine *heiße Violation*. Eine solche verletzt die durch das MSD ausgedrückten Safety-Anforderungen (d. h. ein bestimmtes schädliches Systemverhalten darf nie auftreten) und wird daher auch als *Safety Violation* bezeichnet. Tritt eine heiße Violation in einem System auf, so handelt es sich bei diesem nicht um eine gültige Implementierung der MSD-Spezifikation. Beispielsweise würde ein (erneutes) Auftreten des Nachrichtenereignisses `blankIntact(false)` in der in Abbildung 2.3 gezeigten Situation zu einer heißen Violation im MSD ArmRemovesBrokenBlank führen, da diese Nachricht nicht freigegeben ist und sich das MSD in einem heißen Cut befindet.

Tritt eine Violation jedoch auf, wenn sich das MSD in einem kalten Cut befindet, so handelt es sich um eine *kalte Violation*. Diese verletzt nicht die Anforderungen, sondern führt lediglich zur Beendigung des jeweiligen aktiven MSDs. Dadurch kommen etwaige zusätzliche Anforderungen, die durch dieses MSD ausgedrückt werden, nicht mehr zum Tragen. Beispielsweise führt im MSD ArmMovesBlanksToPress in Abbildung 2.3 ein erneutes Auftreten des Ereignisses `blankToPress` zu einer kalten Violation, wenn sich das MSD in dem eingezeichneten kalten Cut befindet. Dies bewirkt dann die Beendigung des aktiven MSDs für ArmMovesBlanksToPress.

Violations können jedoch nur für Nachrichtenereignisse auftreten, für die eine entsprechende Diagrammnachricht im MSD vorkommt. Ein MSD sagt also nichts über die erlaubten Reihenfolgen von Nachrichten aus, die nicht im MSD vorkommen. Beispielsweise kann das Auftreten der Nachricht `plateToBelt` nie eine Violation im MSD ArmRemovesBrokenBlank bewirken.

Bisher wurden nur Violations beschrieben, die durch das Auftreten eines Nachrichtenereignisses entstehen, das der durch ein MSD vorgegebenen Reihenfolge widerspricht. Es ist jedoch auch eine Violation eines MSDs, wenn dieses für immer in demselben ausgeführten Cut verbleibt. Dies bedeutet für das System, dass in ihm nach Erreichen des ausgeführten Cuts niemals wieder ein Nachrichtenereignis auftritt, das (durch Weiterschalten des Cuts oder eine kalte Violation) zu seinem Verlassen führt. Ein derartiges Systemverhalten wird als *Liveness Violation* („Lebendigkeits“-Verletzung) des MSDs bezeichnet, da es einer Liveness-Anforderung (d. h. ein bestimmtes erforderliches Systemverhalten muss irgendwann passieren) widerspricht. Im MSD ArmRemovesBrokenBlank wäre es beispielsweise eine Liveness Violation, wenn aC1 in dem eingezeichneten Cut niemals `removeBlank` sendet.

Unifizierbarkeit Für die Feststellung, ob ein im System auftretendes Nachrichtenereignis freigegeben ist oder ob es eine Violation verursacht, wird eine Prüfung auf *Unifizierbarkeit* des Ereignisses mit den Diagrammnachrichten der aktiven MSDs durchgeführt.

Dies bedeutet im Wesentlichen, dass geprüft wird, ob das Ereignis in einem bestimmten Sinne der Diagrammnachricht entspricht.

Es werden zwei Grade von Unifizierbarkeit unterschieden: Ein Nachrichtenereignis kann nur dann eine Violation im betreffenden aktiven MSD verursachen, wenn es *Nachrichten-unifizierbar* mit irgendeiner der Diagrammnachrichten ist. Ist dies der Fall, dann wird zusätzlich geprüft, ob das Ereignis mit einer der *freigegebenen* Diagrammnachrichten *Parameter-unifizierbar* ist. Falls auch das gegeben ist, so wird der Cut (also die neue Menge der freigegebenen Nachrichten) entsprechend weitergeschaltet, da die freigegebene Nachricht „aufgetreten“ ist. Ist nur ersteres erfüllt, letzteres aber nicht, so tritt eine Violation auf, die abhängig von der Temperatur des Cuts heiß oder kalt ist.

Ein Ereignis ist dann *Nachrichten-unifizierbar* mit einer Diagrammnachricht, wenn

1. sich Ereignis und Diagrammnachricht auf dieselbe Operation beziehen und
2. das sendende/empfangende Objekt des Ereignisses von der sendenden/empfangenden Lifeline der Diagrammnachricht repräsentiert wird.

Beispielsweise ist in Abbildung 2.3 das erste Ereignis `blankIntact(true)` mit der gleichnamigen minimalen Nachricht des MSDs ArmMovesBlanksToPress Nachrichten-unifizierbar, da die entsprechenden Lifelines des aktiven MSDs an Senderobjekt `s` bzw. Empfängerobjekt aC2 gebunden sind und die Operation in beiden Fällen blankIntact ist. Das später auftretende Ereignis `blankIntact(false)` ist jedoch nicht Nachrichten-unifizierbar mit der minimalen Nachricht, da das Empfängerobjekt in diesem Fall aC1 ist.

Ein Ereignis ist dann *Parameter-unifizierbar*, wenn es Nachrichten-unifizierbar ist und wenn zusätzlich jeder Parameterwert des Ereignisses unifizierbar mit einem entsprechenden durch die Diagrammnachricht definierten Parameterwert ist. Ist letzteres ein konkreter Wert, so muss dieser identisch zum Parameterwert des Ereignisses sein. Der andere Fall, dass die Diagrammnachricht statt eines konkreten Werts eine Variable für den Parameter angibt, wird in Abschnitt 2.1.4 bei der Erläuterung von Variablen in MSDs beschrieben. Hat eine Diagrammnachricht keine Parameter, dann sind alle Ereignisse, die mit dieser Nachricht Nachrichten-unifizierbar sind, auch Parameter-unifizierbar mit ihr.

Im Beispiel in Abbildung 2.3 ist das erste Ereignis `blankIntact(true)` Parameter-unifizierbar mit der minimalen Nachricht des MSDs ArmMovesBlanksToPress, da es Nachrichten-unifizierbar mit dieser ist (siehe oben) und zusätzlich der einzige Parameterwert des Ereignisses identisch zu dem im MSD angegebenen Wert ist. Mit einem Parameterwert `false` wäre das Ereignis mit der minimalen Nachricht nur Nachrichten-unifizierbar.

Enthält ein MSD eine parametrisierte Nachricht, die nicht freigegeben ist, dann führt nach obiger Festlegung jedes Nachrichtenereignis zu einer Violation, das zu dieser Nachricht Nachrichten-unifizierbar ist, auch wenn es zur Diagrammnachricht inkompatible Parameterwerte hat und daher nicht Parameter-unifizierbar mit dieser ist. Definiert die Diagrammnachricht also einen konkreten Parameterwert, wie es in Abbildung 2.3 für die jeweils erste Nachricht in beiden MSDs der Fall ist, dann führt auch ein Ereignis mit einem anderen Wert zu einer Violation.

In dem in der Abbildung gezeigten Cut für das MSD ArmRemovesBrokenBlank würde also beispielsweise ein Auftreten des Ereignisses `blankIntact` mit Sender `s` und Empfänger aC1 eine (heiße) Violation verursachen – auch wenn der Parameter `true` ist.

Die konkreten Anforderungen im Beispiel Durch die beschriebene Semantik modelliert das MSD ArmMovesBlanksToPress in Abbildung 2.3 somit folgende Anforderungen an das Systemverhalten, da andernfalls eine heiße Violation oder eine Liveness Violation auftritt: Der ArmController aC2 muss nach Empfangen der Nachricht `blankIntact` mit dem Parameterwert `true` vom Sensor `s` die Nachricht `blankToPress` an den Arm a2 senden. Dies muss zudem geschehen, bevor aC2 `plateToBelt` an den Arm a2 sendet und bevor `s` erneut `blankIntact` an aC2 sendet.

Die durch das MSD ArmRemovesBrokenBlank ausgedrückte Anforderung ist, dass der ArmController aC1 nach Empfangen der Nachricht `blankIntact` mit dem Parameterwert `false` vom Sensor `s` die Nachricht `removeBlank` an den Arm a1 senden muss. Dies muss geschehen, bevor `s` erneut `blankIntact` an aC1 sendet.

2.1.3 Umgebungsannahmen: Assumption MSDs

Der im vorherigen Abschnitt vorgestellte Formalismus der MSDs kann nicht nur zur Spezifikation von Anforderungen an das zu entwickelnde System, sondern auch zur Formalisierung von Annahmen an dessen Umgebung verwendet werden. Hierfür wird zwischen zwei Arten von MSDs unterschieden: *Requirement MSDs* definieren Anforderungen, während *Assumption MSDs* Annahmen definieren. Zur Unterscheidung der beiden Arten von MSDs werden Assumption MSDs mit dem Stereotyp «Environment-Assumption» gekennzeichnet, welches über dem Namen des MSDs dargestellt wird. Requirement MSDs sind durch das Fehlen dieses Stereotyps erkennbar. Bei den in den Abbildungen 2.2 und 2.3 gezeigten MSDs handelt es sich also um Requirement MSDs. Abbildung 2.4 zeigt ein Beispiel für ein Assumption MSD.

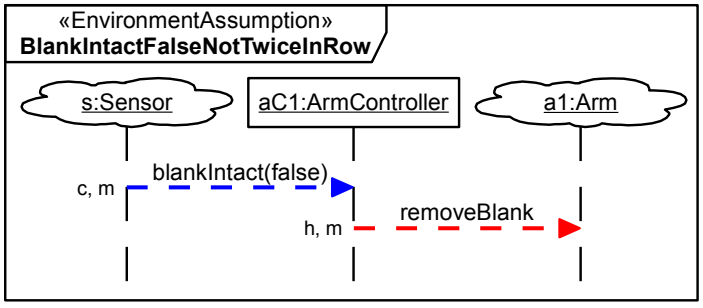


Abbildung 2.4: Beispiel für ein Assumption MSD

Die Syntax der Assumption MSDs ist, abgesehen von ihrer Kennzeichnung durch den Stereotyp, identisch zu der von Requirement MSDs. Semantisch ist die einzige Besonderheit, dass Liveness Violations und Safety Violations von Assumption MSDs nicht Verletzungen der Anforderungen an das System, sondern Verletzungen der Annahmen

an die Umgebung darstellen. Die Umgebung ist quasi „verantwortlich“ dafür, dass die mittels Assumption MSDs modellierten Annahmen eingehalten werden. Ist dies für eine gegebene Umgebung nicht gewährleistet, dann muss auch das System die mittels Requirement MSDs modellierten Anforderungen an dieses nicht einhalten und darf beliebig vom modellierten Verhalten abweichen. Dies bedeutet natürlich auch, dass ein Einsatz des Systems in einer nicht den Annahmen entsprechenden Umgebung nicht sicher ist. Umgekehrt gilt, dass das System die Anforderungen für jedes Umgebungsverhalten erfüllen muss, das nicht durch die Umgebungsannahmen ausgeschlossen wird.

Assumption MSDs werden insbesondere dann benötigt, wenn die Anforderungen an das System von diesem nicht ohne spezielle Annahmen an die Umgebung erfüllt werden können. Beispielsweise sind die beiden in Abbildung 2.3 dargestellten Requirement MSDs nur dann erfüllbar, wenn angenommen wird, dass das System immer schneller ist als die Umgebung und daher `removeBlank` bzw. `blankToPress` immer senden kann, bevor die Umgebung erneut `blankIntact` sendet. Andernfalls können die beiden MSDs nicht erfüllt werden: Beide MSDs werden durch eine Umgebungsnachricht `blankIntact` in einen heißen Cut gezwungen. Im Falle des MSDs ArmRemovesBrokenBlank ist der betreffende Cut in der dargestellten Situation eingezeichnet. Entscheidet sich das Umgebungsobjekt `s`, in dieser Situation erneut die Nachricht `blankIntact` an aC1 zu senden, so tritt eine heiße Violation auf.

Es ist jedoch möglich, dass nur in einzelnen Fällen angenommen werden kann, dass das System vor der nächsten Umgebungsnachricht senden kann. Dann muss diese Annahme explizit durch ein Assumption MSD modelliert werden, um eine realisierbare MSD-Spezifikation zu erhalten. Im speziellen Fall kann das MSD ArmRemovesBrokenBlank nur dann erfüllt werden, wenn angenommen werden kann, dass das Umgebungsobjekt `s` nach jedem Senden von `blankIntact(false)` an aC1 dieselbe Nachricht erst dann wieder an aC1 senden darf, wenn das Requirement MSD den heißen Cut wieder verlassen hat, d. h. nachdem das Systemobjekt aC1 der Klasse ArmController die Nachricht `removeBlank` an den Arm a gesendet hat. Dies wird durch das Assumption MSD BlankIntactFalseNotTwiceInRow in Abbildung 2.4 modelliert.

Um auch das MSD ArmMovesBlanksToPress erfüllen zu können, ist ein entsprechendes zweites Assumption MSD mit ähnlichem Aufbau wie BlankIntactFalseNotTwiceInRow erforderlich. Dieses muss dann das erneute Senden von `blankIntact(true)` an aC2 vor `blankToPress` unterbinden.

2.1.4 Erweiterte Konzepte der Modal Sequence Diagrams

Die bisher beschriebene MSD-Syntax ermöglicht das Definieren von Anforderungen bzw. Annahmen an die nachrichtenbasierte Kommunikation im System. In bestimmten Fällen kann die Modellierung jedoch sehr umständlich sein. Weiterhin kann mit den bisher vorgestellten Mitteln kein Verhalten definiert werden, das von den aktuellen Attributwerten der Objekte des Systems abhängt. Aus diesen Gründen wurden die Syntax und Semantik von LSCs bzw. MSDs um zusätzliche Modellelemente und Konzepte erweitert. Von diesen werden nachfolgend diejenigen erläutert, die für die vorliegende Arbeit relevant sind.

Symbolische Lifelines

Oft sollen Anforderungen oder Annahmen nicht nur für einzelne Objekte, sondern für eine Menge von gleichartigen Objekten gelten. Anstatt für jede betreffende Kombination von Objekten ein eigenes MSD zu erstellen, kann oft ein einziges MSD mit *symbolischen Lifelines* [MHK02] ausreichen. Die bisher vorgestellten Lifelines, die sich immer auf dasselbe Objekt beziehen, werden auch als *konkrete Lifelines* bezeichnet. Symbolische Lifelines beziehen sich für ein gegebenes aktives MSD zur Laufzeit zwar ebenfalls auf jeweils nur ein Objekt, können in anderen aktiven MSDs desselben MSDs aber andere Objekte referenzieren.

Graphisch unterscheidet sich eine symbolische Lifeline von einer statischen Lifeline darin, dass der Text im Rechteck (bzw. der Wolke) nicht unterstrichen ist. Abbildung 2.5 zeigt das MSD NewBlankNotTwice, das eine symbolische Lifeline mit dem Text `aC:ArmController` enthält. Bei symbolischen Lifelines bezeichnet der Text vor dem Semikolon (im Beispiel: `aC`) nicht das referenzierte Objekt, sondern er definiert den Namen der Lifeline selbst. Der Text nach dem Semikolon (im Beispiel: `ArmController`) gibt eine Klasse an, die der Lifeline zugeordnet werden soll. Eine symbolische Lifeline kann nur Instanzen der ihr zugeordneten Klasse repräsentieren. Im konkreten Beispiel betrifft das MSD also die Kommunikation von Sensor s zu nicht nur einem ArmController, sondern zu beiden im Beispiel-System vorhandenen.

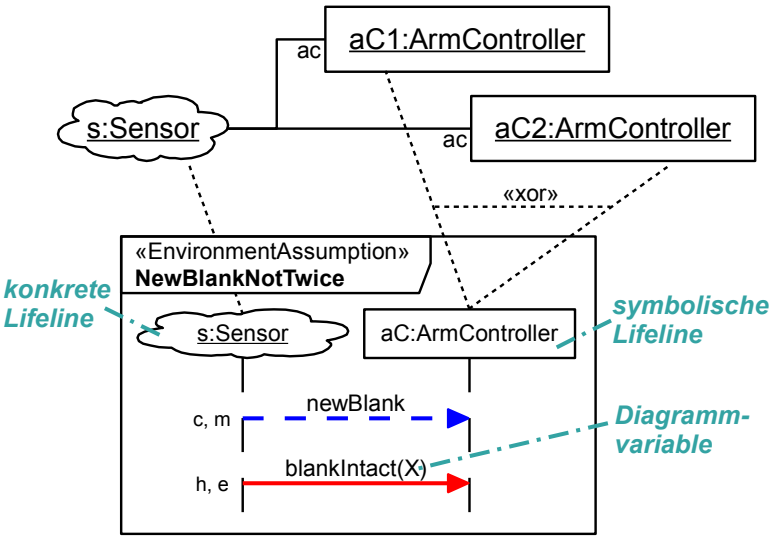


Abbildung 2.5: Beispiel für die Verwendung einer symbolischen Lifeline in einem MSD (unten) und Objektdiagramm mit möglichen Lifeline Bindings (oben)

Die Festlegung, welches konkrete Objekt durch eine symbolische Lifeline repräsentiert wird, erfolgt zur Laufzeit durch das *Binden* der Lifeline an das Objekt. Ergebnis des Bindens ist ein Lifeline Binding, also eine Zuordnung der Lifeline zum Objekt, wie sie bei konkreten Lifelines bereits zur Entwurfszeit gegeben ist. Eine Lifeline, für die in einem aktiven MSD noch kein Lifeline Binding festgelegt wurde, wird als *ungebunden*

bezeichnet. Aufgrund dieser Möglichkeit des Auftretens ungebundener Lifelines ist bei Verwendung von symbolischen Lifelines die Unifizierbarkeit von Diagrammnachrichten und Nachrichtenereignissen abweichend zur Beschreibung in Abschnitt 2.1.2 definiert (Änderung hervorgehoben):

Ein Ereignis ist dann *Nachrichten-unifizierbar* mit einer Diagrammnachricht, wenn

1. sich Ereignis und Diagrammnachricht auf dieselbe Operation beziehen und
2. wenn für das sendende/empfangende Objekt des Ereignisses und die sendende/empfangende Lifeline der Diagrammnachricht gilt:
   a) Die Lifeline ist an das Objekt gebunden *oder*
   b) *die Lifeline ist eine ungebundene symbolische Lifeline und das Objekt ist eine Instanz der Klasse, die der Lifeline zugeordnet ist.*

Die Definition der Parameter-Unifizierbarkeit an sich bleibt identisch, bezieht sich aber nun auf die Nachrichten-Unifizierbarkeit gemäß dieser neuen Definition.

Beim Binden einer Lifeline können zwei Fälle unterschieden werden. Der eine Fall ist, dass die Lifeline die Sender- oder Empfänger-Lifeline der minimalen Nachricht eines MSDs ist. Dann wird diese beim Auftreten eines mit der minimalen Nachricht Parameter-unifizierbaren Ereignisses an das jeweilige Sender- bzw. Empfängerobjekt gebunden. Der beschriebene Fall trifft für die symbolische Lifeline aC in Abbildung 2.5 zu, da diese die minimale Nachricht `newBlank` empfängt. Da die Objekte aC1 und aC2 Instanzen der Klasse ArmController sind, sind von s gesendete `newBlank`-Ereignisse mit Empfängerobjekt aC1 oder aC2 mit der minimalen Diagrammnachricht Parameter-unifizierbar. Somit wird die Lifeline aC bei Auftreten eines solchen Ereignisses an den Empfänger, also entweder aC1 oder aC2 gebunden (siehe Abbildung 2.5). Insgesamt definiert das MSD die Anforderung, dass der Sensor s nach Senden von `newBlank` an einen der ArmController diese Nachricht nicht erneut an denselben Empfänger senden darf, bis er die Nachricht `blankIntact` an diesen gesendet hat. Durch die Angabe einer *Diagrammvariablen* X als Parameter wird hier ausgedrückt, dass die Nachricht einen beliebigen Parameter haben kann. Diagrammvariablen werden in einem späteren Abschnitt näher erläutert.

Im anderen Fall, also wenn die Lifeline nicht Sender- oder Empfänger-Lifeline der minimalen Nachricht ist, dann muss für die Lifeline eine *Binding Expression* angegeben werden. Eine Binding Expression ist ein UML-Kommentar, der einen Ausdruck in der *Object Constraint Language (OCL)* [Obj10] enthält und der jeweiligen Lifeline zugeordnet ist. Abbildung 2.7 zeigt ein Beispiel, in dem der Lifeline a eine Binding Expression zugeordnet ist. Die Auswertung des OCL-Ausdrucks der Binding Expression muss genau ein Objekt ergeben, das eine Instanz der Klasse ist, die der Lifeline zugeordnet ist. Im Beispiel muss es also vom Typ Arm sein.

Binding Expressions können auf *Lifeline-Variablen* zugreifen. Lifeline-Variablen repräsentieren das Objekt, an das eine Lifeline gebunden ist und ermöglichen den Zugriff auf dessen Attribute bzw. Assoziationen. Im Beispiel wird mittels „aC.arm“ auf das Arm-Objekt zugegriffen, das (über die „arm“-Assoziation) demjenigen ArmController zugeordnet ist, der von aC repräsentiert wird. Im Beispiel ist das a1 für aC1 und a2 für aC2. Alle Binding Expressions eines MSDs werden unmittelbar nach Aktivierung des

MSDs ausgewertet. Im Beispiel ist aC zu diesem Zeitpunkt gebunden, da diese Lifeline die minimale Nachricht des MSD empfängt. Die restliche Bedeutung des Beispiel-MSDs wird im nächsten Abschnitt beschrieben.

Lifeline Bindings können nach ihrer Erzeugung nicht mehr geändert werden. Sie bleiben gültig, solange das aktive MSD besteht, werden mit dessen Beendigung jedoch ebenfalls entfernt. Sie gelten nur für ein einziges aktives MSD und haben keine Auswirkungen auf andere aktive MSDs, die dasselbe MSD instanziieren. Generell werden zwei aktive MSDs eines gegebenen MSDs als unterschiedlich angesehen, wenn sich ihr Cut, ihre Lifeline Bindings, oder beide unterscheiden.

Variablen und Variablenzugriffe

Um Verhalten abhängig von den aktuellen Attributwerten von Objekten zu definieren oder Nachrichtenparameter zueinander in Bezug setzen zu können, wurden MSDs um verschiedene Typen von *Variablen* erweitert [MHK02]. Das Konstrukt der *Bedingung* ermöglicht es, Verhalten in MSDs nur bei Vorliegen bestimmter Variablenwerte auszuführen. Um Variablenwerte zur Laufzeit verändern zu können, wurde weiterhin das Konstrukt der *Zuweisung* in MSDs eingeführt. Die drei MSDs in Abbildung 2.6 illustrieren die Verwendung von Variablen, Bedingungen und Zuweisungen zur Modellierung einer Beispielanforderung und einer zugehörigen Annahme.

Zusammengenommen modellieren die MSDs in Abbildung 2.6 in erster Linie die zusätzliche Funktionalität, dass das System an das Benutzerinterface (die UI ui) der Produktionsanlage eine Warnung sendet, wenn drei defekte Rohlinge nacheinander ankommen, also ohne Unterbrechung durch einen intakten Rohling. Diese Anforderung soll dabei durch den ArmController aC1 umgesetzt werden, der normalerweise aber nur Meldungen über defekte Rohlinge erhält (siehe Abbildung 2.3).

Für das Erfüllen dieser Anforderung wird daher die durch das Assumption MSD AC1getsAllSensorMsgs modellierte Annahme benötigt, dass der Sensor s alle Nachrichten, die er an aC2 sendet, mit demselben Parameter auch an aC1 sendet. Die heiße ausgeführte Diagrammnachricht stellt sicher, dass s nach jedem Senden von `blankIntact` an aC2 irgendwann eine Kopie an aC1 sendet und vorher keine weitere `blankIntact`-Nachricht an aC2 sendet.

Um sicherzustellen, dass der Parameter der Kopie derselbe ist wie beim Original, wird im MSD AC1getsAllSensorMsgs die *Diagrammvariable* mit dem Bezeichner X verwendet. Diese ist zunächst *ungebunden*, d. h. sie hat noch keinen speziellen Wert. Variablen mit einem konkreten Wert werden als *gebunden* bezeichnet. Wenn ein MSD für den Parameter einer Nachricht statt eines konkreten Werts eine Diagrammvariable definiert, dann ändern sich die Kriterien für Parameter-Unifizierbarkeit wie folgt (Änderungen hervorgehoben):

Ein Ereignis ist dann *Parameter-unifizierbar*, wenn es Nachrichten-unifizierbar ist und wenn zusätzlich jeder Parameterwert des Ereignisses unifizierbar mit dem durch die Diagrammnachricht definierten Parameterwert ist. Ist letzteres ein konkreter Wert *oder eine gebundene Variable*, so muss dieser *bzw. der Variablenwert* dazu identisch zum Parameterwert des Ereignisses sein. *Ist der Parameterwert der Diagrammnachricht eine ungebundene Variable, dann kann der Parameterwert des Ereignisses beliebig sein.*

Wenn ein Nachrichtenereignis auftritt, das mit einer Diagrammnachricht Parameter-unifizierbar ist, dann werden sämtliche ungebundenen Diagrammvariablen, welche für die Parameter der Diagrammnachricht stehen, an den entsprechenden Parameterwert dieses Nachrichtenereignisses *gebunden*. Für die betroffenen aktiven MSDs hat die Variable ab dem Zeitpunkt des Bindens diesen Wert, bis das jeweilige aktive MSD beendet wird oder der Variablen ein neuer Wert zugewiesen wird. Tritt also im Beispiel ein Ereignis `blankIntact(true)` von `s` an `aC2` auf, dann erhält `X` in `AC1getsAllSensorMsgs` den Wert `true`. Dementsprechend kann die Umgebung das aktive Assumption MSD ab diesem Zeitpunkt nur noch durch Senden von `blankIntact(true)` von `s` an `aC1` erfüllen, nicht mehr durch `blankIntact(false)`.

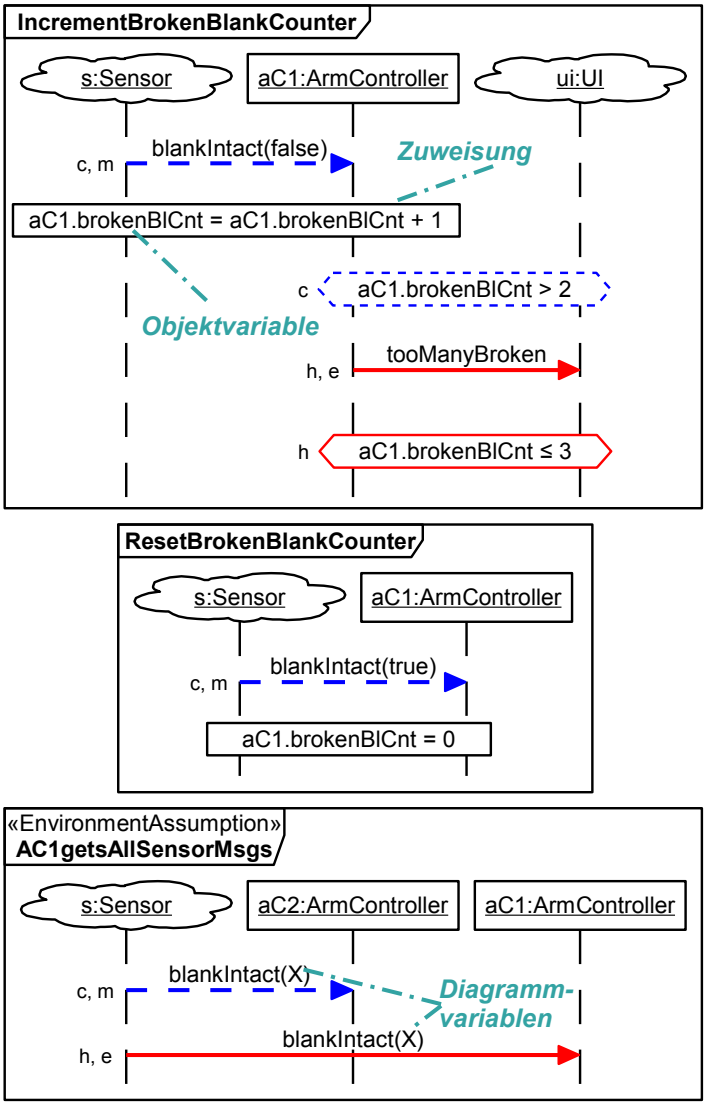


Abbildung 2.6: Beispiel für Variablen, Bedingungen und Zuweisungen in MSDs

Eine Bindung und damit der Wert einer Diagrammvariablen gilt jedoch jeweils nur lokal für das betroffene aktive MSD. Es ist also nicht möglich, aus einem aktiven MSD heraus auf die Werte der Diagrammvariablen anderer aktiver MSDs zuzugreifen, auch wenn diese Variablen gleich benannt sind oder sie sogar durch dasselbe MSD definiert wurden.

Um die oben erwähnte Funktionalität einer Warnung bei zu vielen defekten Rohlingen umzusetzen, müssen die auftretenden defekten Rohlinge zunächst gezählt werden. Die Klasse ArmController modelliert dazu die Anzahl bisher aufgetretener defekter Rohlinge durch das Integer-Attribut brokenBlCnt (vgl. das Klassendiagramm in Abbildung 2.1). Das Requirement MSD IncrementBrokenBlankCounter greift auf dieses Attribut in Form der *Objektvariablen* aC1.brokenBlCnt zu. Eine Objektvariable repräsentiert ein Attribut eines an eine Lifeline gebundenen Objekts, in diesem Fall also das Attribut brokenBlCnt des Objekts aC1. Objektvariablen sind nie ungebunden, sondern haben einen Default-Wert abhängig von ihrem Typ. Im Falle von Integer-Variablen wie brokenBlCnt ist dies der Wert `0`.

Bei jedem Auftreten einer Nachricht `blankIntact(false)` an aC1, also bei jedem defekten Rohling, wird das MSD IncrementBrokenBlankCounter instanziiert. Direkt unterhalb der minimalen Nachricht befindet sich im MSD eine *Zuweisung*. Eine Zuweisung wird durch ein weißes Rechteck mit schwarzem Rand repräsentiert, das einen Text der Form `<Variablenname> = <OCL-Ausdruck>` enthält. Der Text `<Variablenname>` kann dabei entweder eine Diagrammvariable oder eine Objektvariable bezeichnen. Der Text `<OCL-Ausdruck>` ist ein OCL-Ausdruck, dessen Ergebnis vom Typ dieser Variable sein muss. Hat der Cut eine Zuweisung auf allen Lifelines erreicht, so wird diese sofort (also insbesondere vor dem Auftreten weiterer Nachrichtenereignisse) ausgeführt und der Cut erreicht die jeweils auf den Lifelines nachfolgenden Diagrammelemente. Bei der Ausführung wird der Variablen der Ergebniswert des OCL-Ausdrucks zugewiesen. Im konkreten Fall im Beispiel-MSD wird dadurch der Wert der Objektvariablen aC1.brokenBlCnt inkrementiert.

Das MSD IncrementBrokenBlankCounter berücksichtigt jedoch nicht, dass die Variable aC1.brokenBlCnt nicht alle defekten Rohlinge seit Systemstart zählen soll, sondern nur diejenigen, die seit dem letzten intakten Rohling angekommen sind. Die Integer-Variable aC1.brokenBlCnt muss also bei Auftreten jedes intakten Rohlings auf `0` gesetzt werden. Das MSD ResetBrokenBlankCounter in Abbildung 2.6 modelliert dies ebenfalls durch eine entsprechende Zuweisung.

Das Senden der Warnnachricht selbst wird ebenfalls durch das MSD IncrementBrokenBlankCounter modelliert. Da die Nachricht bei Auftreten von mindestens drei defekten Rohlingen in Folge gesendet werden soll, wird die Objektvariable aC1.brokenBlCnt auf einen entsprechenden Wert überprüft. Dies erfolgt durch eine *Bedingung*. Eine Bedingung wird graphisch durch ein Sechseck repräsentiert, welches einen OCL-Ausdruck beinhaltet, der einen Boolean-Wert zurückgibt. Die Bedingung gilt genau dann als erfüllt, wenn der OCL-Ausdruck zu `true` ausgewertet wird. Im konkreten Beispiel muss dazu also der Ausdruck `aC1.brokenBlCnt > 2` erfüllt sein. Die Bedingung wird erst dann überprüft, wenn der Cut diese auf *allen* Lifelines erreicht hat, die von der Bedingung überdeckt werden. Dann aber findet die Überprüfung sofort statt, bevor weitere Ereignisse im System auftreten können. Ist die Bedingung erfüllt, so werden die nächsten Elemente

unterhalb der Bedingung freigegeben, was den Cut entsprechend verändert. Im Beispiel wird dann die Diagrammnachricht `tooManyBroken` freigegeben, die durch das System vor dem Ankommen des nächsten defekten Rohlings gesendet werden muss.

Ist eine freigegebene Bedingung nicht erfüllt, dann hängt die Auswirkung von der *Temperatur* der Bedingung ab. Wie bei einer Nachricht kann die Temperatur einer Bedingung entweder *heiß* oder *kalt* sein. Eine Bedingung hat jedoch keine Ausführungsart. Graphisch wird die Temperatur durch die Farbe und Art des sechseckigen Rahmens der Bedingung repräsentiert: Eine heiße Bedingung hat einen durchgezogenen roten Rahmen, während dieser bei einer kalten Bedingung gestrichelt und blau ist. Zusätzlich wird die Temperatur textuell neben der Bedingung durch h für heiß (engl. ***h**ot*) oder c für kalt (engl. ***c**old*) angegeben. Die Bedingung `aC1.brokenBlCnt > 2` im MSD IncrementBrokenBlankCounter ist also kalt. Ist eine kalte Bedingung nicht erfüllt, wenn sie ausgewertet wird, so wird das aktive MSD sofort beendet. Im Beispiel wird der Cut also bei `aC1.brokenBlCnt` $\leq$ `2` die Diagrammnachricht `tooManyBroken` nie erreichen.

Ist im MSD IncrementBrokenBlankCounter die kalte Bedingung erfüllt und wurde anschließend die Nachricht `tooManyBroken` gesendet, so wird eine zweite Bedingung `aC1.brokenBlCnt` $\leq$ `3` erreicht, die jedoch heiß ist. Ist eine heiße Bedingung bei ihrer Auswertung nicht erfüllt, so bleibt sie freigegeben und der Cut ändert sich nicht. Bei Änderung der im Ausdruck vorkommenden Variablen wird sie erneut überprüft. Sie bleibt solange freigegeben, bis sie irgendwann erfüllt ist oder das aktive MSD aus anderen Gründen beendet wird. Solange eine heiße Bedingung freigegeben bleibt, ist das aktive MSD in einem heißen ausgeführten Cut. Es darf also keine andere Nachricht des MSD auftreten und es ist eine Liveness Violation, wenn die Bedingung nie erfüllt wird. Im konkreten Beispiel ist die Bedingung bei `aC1.brokenBlCnt > 3` nicht erfüllt. Da ein danach auftretendes `blankIntact(true)` wegen des heißen Cuts eine heiße Violation verursachen würde, kann die Bedingung nicht mehr erfüllt werden, was wiederum zu einer Liveness Violation führt. Die durch die Bedingung modellierte Anforderung an das System ist also, dass niemals mehr als drei defekte Rohlinge in Folge auftreten dürfen. Ist `aC1.brokenBlCnt` $\leq$ `3` jedoch erfüllt, so erreicht das MSD sein Ende und terminiert.

Verbotene Nachrichten

In der Praxis sind oft Annahmen oder Anforderungen zu modellieren, in denen eine bestimmte Nachricht in einer bestimmten Situation entweder immer oder nie erlaubt ist. Dies kann mit der bisher beschriebenen Syntax nicht explizit modelliert werden, da es von der Temperatur des aktuellen Cuts abhängt, ob eine Violation heiß oder kalt ist. Daher wurde das *Forbidden-Fragment* in MSDs eingeführt. Dieses wird graphisch durch ein Rechteck repräsentiert, das mit dem Schriftzug „forbidden“ gekennzeichnet ist.

Verbotene Nachrichten haben keine Ausführungsart und werden immer mit gestricheltem Pfeil dargestellt. Sie haben aber, wie gewöhnliche Diagrammnachrichten, eine Temperatur, die heiß oder kalt sein kann. Graphisch werden sie dementsprechend durch rote bzw. blaue Pfeile repräsentiert und mit h bzw. c markiert.

Die Semantik der Temperatur verbotener Nachrichten ist jedoch im Unterschied zu anderen Diagrammnachrichten, dass sie festlegt, welche Art von Violation diese Nachrichten auslösen. Heiße verbotene Nachrichten lösen immer eine heiße Violation aus,

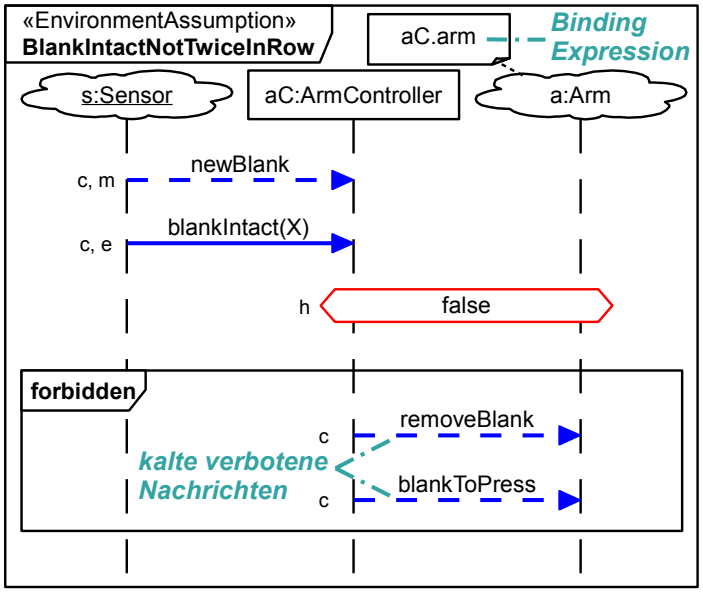


Abbildung 2.7: Beispiel für die Verwendung eines Forbidden-Fragments

kalte verbotene Nachrichten eine kalte Violation. Dies gilt so lange das aktive MSD existiert, aber unabhängig vom aktuellen Cut; die Festlegung durch das Forbidden-Fragment hat also Vorrang.

Das Assumption MSD BlankIntactNotTwiceInRow in Abbildung 2.7 modelliert, dass nach den Nachrichten `newBlank` und `blankIntact` eine heiße Bedingung `false` erreicht wird, die niemals erfüllt werden kann. Der Cut verbleibt also in einem heißen ausgeführten Cut vor der Bedingung. Im Forbidden-Fragment des MSDs sind jedoch zwei kalte verbotene Nachrichten definiert. Diese führen selbst in einem heißen Cut zu einer kalten Violation. Daher bewirkt die heiße Bedingung hier zwar, dass der Sensor `s` nicht erneut `newBlank` und `blankIntact` an dasselbe Objekt senden darf; die kalten verbotenen Nachrichten `removeBlank` und `blankToPress` sind jedoch weiterhin möglich. Ihr Auftreten bewirkt, dass das aktive MSD beendet wird und `s` wieder `newBlank` oder `blankIntact` senden darf.

Insgesamt ersetzt das MSD BlankIntactNotTwiceInRow das speziell auf aC1 bezogene Assumption MSD BlankIntactFalseNotTwiceInRow in Abbildung 2.4, sowie eine entsprechende Variante für den zweiten ArmController aC2 und garantiert zusätzlich, dass der Sensor `s` nach `newBlank` auf jeden Fall `blankIntact` sendet.

Erreicht der Cut auf allen Lifelines ein Forbidden-Fragment, so wird das aktive MSD sofort beendet. Die darin definierten verbotenen Nachrichten werden niemals vom Cut erreicht und daher niemals freigegeben. Auch ihre Reihenfolge auf den jeweiligen Lifelines ist unerheblich.

2.1.5 Echtzeitanforderungen und -annahmen: Timed MSDs

Mechatronische Systeme müssen für ein korrektes Funktionieren oft nicht nur für alle Eingaben die korrekten Nachrichten senden, sondern sie müssen dies auch zu den richtigen Zeiten tun. Beispielsweise kann es beim beschriebenen Produktionssystem passieren, dass der Roboterarm a2 durch die Presse zerstört wird, wenn er zu früh die Anweisung erhält, die gepresste Metallplatte zum Ablageförderband zu bewegen. Um dies zu verhindern, kann als Anforderung aufgenommen werden, dass nach dem Bewegen des Rohlings zur Presse eine bestimmte Zeitspanne vergehen muss, bis das Steuergerät des Roboterarms den nächsten Befehl übermittelt.

Um derartige Anforderungen und Annahmen ausdrücken zu können, wurden MSDs um Modellelemente erweitert, die es ermöglichen, über die erlaubten zeitlichen Abstände zwischen Nachrichten bzw. Nachrichtensequenzen Aussagen zu treffen [HM02b]. Diese Modellelemente werden in diesem Abschnitt beschrieben. Die derart erweiterten MSDs werden in dieser Arbeit als *zeitbehaftete MSDs* bezeichnet.

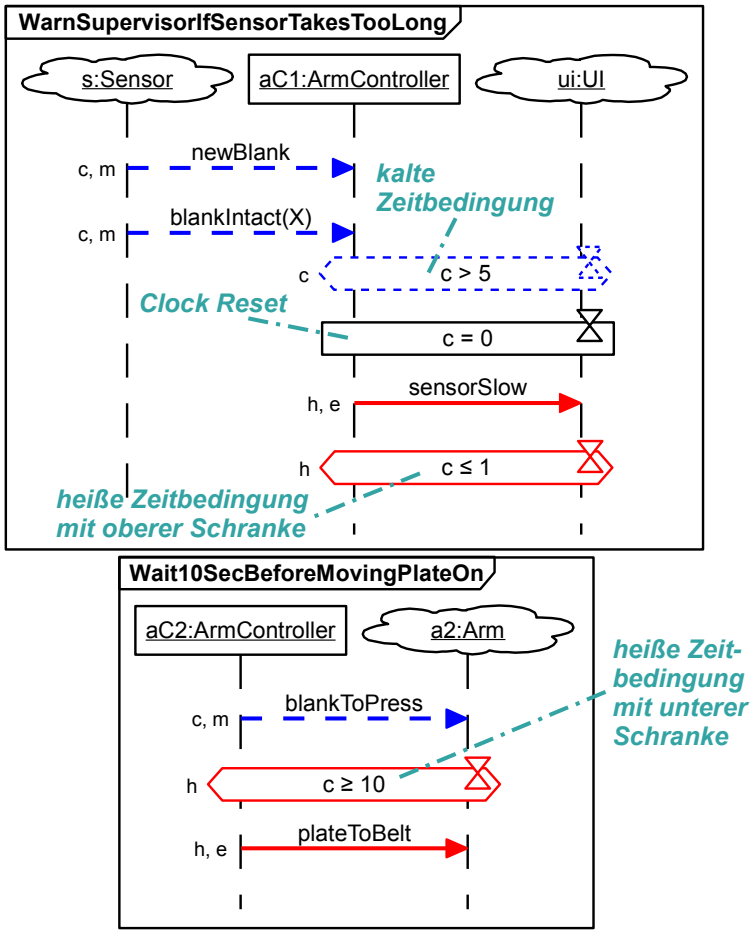


Abbildung 2.8: Beispiel für Echtzeitanforderungen und Timed MSDs

Das Requirement MSD Wait10SecBeforeMovingPlateOn in Abbildung 2.8 modelliert die Anforderung, dass der ArmController aC2 nach Senden von `blankToPress` mindestens zehn Zeiteinheiten warten muss, bis er `plateToBelt` sendet. Die konkrete Dauer einer Zeiteinheit kann für jedes System einzeln festgelegt werden. Im Rahmen dieser Arbeit wird sie als eine Sekunde definiert. Die Anforderung wird im MSD durch eine *Zeitbedingung* modelliert.

Eine Zeitbedingung wird graphisch wie eine (normale) Bedingung durch ein Sechseck repräsentiert, mit dem Unterschied, dass an der rechten oberen Ecke des Sechsecks eine stilisierte Sanduhr dargestellt ist. Zeitbedingungen haben ebenfalls eine Temperatur, die heiß oder kalt sein kann, was graphisch durch einen blau gestrichelten (für kalt) bzw. rot durchgezogenen (für heiß) Rand dargestellt wird. Die Bedingung in Wait10SecBeforeMovingPlateOn ist also heiß.

Bei Zeitbedingungen enthält das Sechseck statt einer OCL-Bedingung eine Konjunktion von *Uhrenvergleichen*. Diese haben die Form `<Uhr> <Operator> <Uhrwert>`. Dabei gibt `<Uhr>` den Namen einer *Uhr* an. Uhren sind spezielle Variablen, die wie Diagrammvariablen durch Verwendung im MSD deklariert werden. Sie sind jedoch niemals ungebunden, sondern sie werden instanziiert, wenn das definierende MSD instanziiert wird, d. h. wenn ein aktives MSD zu diesem erzeugt wird. Uhren sind immer kontinuierliche Variablen (Wertebereich $\mathbb{R}$), die initial einen Wert von `0` haben und beim Vergehen von Zeit kontinuierlich mit konstanter Rate im Wert steigen. Dabei wird angenommen, dass diese Rate bei allen Uhren im System identisch ist. Alle Uhren laufen also immer synchron.

Der Wert `<Uhrwert>` ist ein Wert aus $\mathbb{N}$, mit dem der aktuelle Wert von `<Uhr>` verglichen wird. Die Art dieses Vergleichs wird dabei durch den Vergleichsoperator `<Operator>` $\in \{<, \leq, =, \geq, >\}$ definiert.

Ein einzelner Uhrenvergleich hat genau dann den Wert `true`, wenn er für den aktuellen Wert der jeweiligen Uhr erfüllt ist, andernfalls `false`. Eine Zeitbedingung ist somit genau dann erfüllt, wenn ihre Formel für die aktuellen Werte aller darin verglichenen Uhren erfüllt ist. Die Zeitbedingung im MSD Wait10SecBeforeMovingPlateOn in Abbildung 2.8 ist also erfüllt, wenn der Wert der Uhr `c` mindestens 10 ist, also mindestens diese Zeitspanne seit Erzeugung des aktiven MSDs durch die Nachricht `blankToPress` vergangen ist.

Wie eine Bedingung wird eine Zeitbedingung sofort ausgewertet, sobald sie freigegeben ist, d. h. wenn der Cut diese auf allen Lifelines erreicht hat. Die Auswirkungen einer erfüllten bzw. nicht erfüllten Zeitbedingung sind dieselben wie bei Bedingungen. Bei erfüllten Zeitbedingungen springt der Cut also direkt hinter diese, bei nicht erfüllten kalten Zeitbedingungen tritt eine kalte Violation auf und bei nicht erfüllten heißen Zeitbedingungen verbleibt das aktive MSD in demselben heißen ausgeführten Cut. Die Zeitbedingung `c` $\geq$ `10` im MSD Wait10SecBeforeMovingPlateOn in Abbildung 2.8 bewirkt also, dass nach dem Senden von `blankToPress` für 10 Sekunden die Nachrichten `blankToPress` und `plateToBelt` nicht gesendet werden dürfen, da sie aufgrund des heißen Cuts eine heiße Violation verursachen würden. Eine Liveness Violation ist hier ausgeschlossen, da die Uhr `c` irgendwann den Wert 10 erreichen muss.

Das Requirement MSD WarnSupervisorIfSensorTakesTooLong in Abbildung 2.8 modelliert die Anforderung, dass der ArmController aC1 eine Warnung (die Nachricht `sensor-`

Slow) an das Benutzerinterface (die UI ui) senden muss, wenn der Sensor s nach Senden von newBlank mehr als 5 Sekunden benötigt, bis er blankIntact sendet. Hierfür wird eine kalte Zeitbedingung c > 5 verwendet, die nach Empfang der ersten beiden Nachrichten das aktive MSD durch eine kalte Violation beendet, wenn *höchstens* 5 Sekunden zwischen diesen vergangen sind. Andernfalls muss das Systemobjekt aC1 die Nachricht sensorSlow senden, was durch eine entsprechende heiße ausgeführte Diagrammnachricht modelliert wird.

Unmittelbar nach der kalten Zeitbedingung erreicht der Cut einen *Clock Reset*. Bei diesem handelt es sich um eine spezielle Art von Zuweisung, bei dessen Erreichen einer Uhr ein Wert von 0 zugewiesen wird. Der Text hat daher für eine Uhr mit dem Namen <Uhr> immer die Form <Uhr> = 0. Graphisch unterscheidet sich ein Clock Reset von einer gewöhnlichen Zuweisung nur durch eine stilisierte Sanduhr in der rechten oberen Ecke des Rechtecks.

Wie bei normalen Zuweisungen werden Clock Resets sofort bei Freigabe ausgeführt und die nachfolgenden Elemente werden freigegeben bevor weitere Ereignisse auftreten können. Clock Resets ermöglichen es, die vergangene Zeit seit dem Senden anderer Nachrichten als der minimalen Diagrammnachricht zu modellieren.

Im Beispiel in Abbildung 2.8 modelliert die Uhr c im MSD WarnSupervisorIfSensorTakesTooLong ab dem Clock Reset die Zeit seit dem Auftreten des Nachrichtenereignisses blankIntact, da die Zeitbedingung und der Clock Reset unmittelbar nach diesem Ereignis ausgeführt werden und selbst keine Zeit benötigen. Die so zurückgesetzte Uhr wird anschließend in einer heißen Zeitbedingung c ≤ 1 nach der Diagrammnachricht sensorSlow überprüft. Sendet der ArmController aC1 die Nachricht sensorSlow also innerhalb einer Sekunde nach der Nachricht blankIntact, so ist die Zeitbedingung erfüllt und das aktive MSD wird beendet. Andernfalls kann die Zeitbedingung niemals mehr erfüllt werden, da die Uhr c bereits einen größeren Wert als 1 hat und dieser nur noch steigen, aber nicht mehr sinken kann. Dadurch modelliert das MSD die zusätzliche Anforderung, dass aC1 die Nachricht sensorSlow nicht irgendwann, sondern innerhalb der vorgegebenen Zeitspanne senden muss.

Zur Umsetzung zeitbehafteter Anforderungen sind oft ebenfalls zeitbehaftete Annahmen erforderlich. Beispielsweise reicht es nicht aus, durch die oben beschriebene verzögerte Ansteuerung des Roboterarms der Presse eine Zeitspanne von zehn Sekunden einzuräumen, wenn der Pressvorgang tatsächlich länger als diesen Zeitraum dauert. In diesem einfachen Beispiel wurde die Presse nicht modelliert; nimmt man diese jedoch in die MSD-Spezifikation auf, so kann man die tatsächlich benötigte Zeit des Pressvorgangs durch Verwendung der hier beschriebenen Modellelemente in entsprechenden Assumption MSDs modellieren. Das Syntheseverfahren kann diese Annahmen dann berücksichtigen und bei der Erzeugung des Systemverhaltens sicherstellen, dass dieses die zur Verfügung stehende Zeitdauer nicht überschreitet. In Kapitel 4 wird eine Variante des Beispiels mit Echtzeitanforderungen und -annahmen vorgestellt.

2.1.6 Implementierung einer MSD-Spezifikation

In Abschnitt 2.1.2 wurde bereits erläutert, dass ein einzelnes MSD verletzt wird, wenn in diesem eine heiße Violation (auch: Safety Violation) oder eine Liveness Violation

auftritt. Dieser Abschnitt befasst sich nun mit der Frage, wann ein System gemäß einer MSD-Spezifikation korrekt ist, also wann es sie erfüllt.

Sofern das Umgebungsverhalten bei der Anforderungsmodellierung eines Systems berücksichtigt wird, bezeichnet man dieses als *offen*, andernfalls als *geschlossen* [BH05]. Im Rahmen dieser Arbeit wird der allgemeinere Fall der offenen Systeme betrachtet. Das Verhalten der Umgebung muss deshalb bei der Frage berücksichtigt werden, ob die Implementierung korrekt ist.

Damit eine MSD-Spezifikation überhaupt implementiert werden kann, muss diese *konsistent* sein. Dies bedeutet, dass die durch die Requirement MSDs spezifizierten Anforderungen sich (bei gleichzeitig erfüllten Assumption MSDs) nicht widersprechen dürfen. Ist dies der Fall, so existiert eine Implementierung der MSD-Spezifikation, andernfalls nicht [Gre11].

Nachfolgend werden zunächst die Kriterien für eine korrekte Implementierung vorgestellt, die für den in Kapitel 3 vorgestellten Syntheseansatz vorausgesetzt werden. Diese müssen jedoch zur Berücksichtigung von Echtzeitverhalten und asynchroner Kommunikation geringfügig verändert werden. Daher werden für das entsprechend erweiterte Syntheseverfahren in Abschnitt 4 andere Kriterien vorausgesetzt, die hier ebenfalls vorgestellt werden. Schließlich werden implizite Annahmen an die Umgebung behandelt, die in dieser Arbeit gelten.

Allgemeine Kriterien für korrekte Implementierungen

Eine *Implementierung* einer MSD-Spezifikation ist dann korrekt, wenn sie in Kombination mit ihrer Umgebung nur *Abläufe* (siehe Abschnitt 2.1.2) beschreibt, welche die MSD-Spezifikation *erfüllen*. Ein Ablauf erfüllt eine MSD-Spezifikation, wenn dieser entweder von allen Requirement MSDs *akzeptiert* wird, oder er von mindestens einem Assumption MSD nicht akzeptiert wird. Ein MSD akzeptiert einen Ablauf, wenn in diesem Ablauf keine heißen Violations und keine Liveness Violations dieses MSDs auftreten.

Sofern eine MSD-Spezifikation Umgebungsannahmen (also Assumption MSDs) enthält, so müssen demnach die Anforderungen (also die Requirement MSDs) nur bei Einhaltung dieser Annahmen erfüllt werden. Das System muss jegliches Umgebungsverhalten, das den Annahmen widerspricht, also nicht berücksichtigen. Es darf sich, sofern es trotzdem auftritt, beliebig verhalten. Dies heißt aber auch, dass das System nur in Umgebungen eingesetzt werden darf, die den Annahmen genügen.

Im Rahmen dieser Arbeit wird die Bedingung, dass in keinem Requirement MSD eine Liveness Violation auftreten darf, durch eine einfacher zu prüfende Bedingung ersetzt, die erstere impliziert [Gre11]: Das System muss immer wieder einen Zustand erreichen, in dem für *alle* Requirement MSDs gilt, dass *kein* ausgeführter Cut aktiv ist. Diese Bedingung ist restriktiver, da nach ihr Implementierungen inkorrekt sind, bei denen zwar keine Violations in Requirement MSDs auftreten, aber es immer *irgendein* Requirement MSD gibt, das sich in einem ausgeführten Cut befindet.

Diese geänderte Bedingung gilt analog auch für Assumption MSDs, d. h. es verletzt die Umgebungsannahmen, wenn sich immer irgendein Assumption MSD in einem ausgeführten Cut befindet.

Kriterien für korrekte zeitbehaftete Implementierungen

Für zeitbehaftete Implementierungen von (zeitbehafteten) MSD-Spezifikationen gelten die bisherigen Ausführungen im Wesentlichen ebenfalls. Generell sind bei zeitbehafteten Systemen jedoch auch die zeitlichen Abstände zwischen Nachrichtenereignissen für Abläufe des Systems relevant: Durch Verwendung zeitbehafteter Modellelemente (siehe Abschnitt 2.1.5) in Requirement MSDs oder Assumption MSDs können diese dieselbe Folge von Nachrichtenereignissen in einigen Fällen akzeptieren, in anderen Fällen aber aufgrund anderer Zeitabstände nicht akzeptieren. Für zeitbehaftete Systeme und zeitbehaftete MSD Spezifikationen werden Folgen von Nachrichtenereignissen mit unterschiedlichen Zeitabständen daher jeweils als unterschiedliche Abläufe betrachtet.

Implizite Annahmen an das Umgebungsverhalten

In dieser Arbeit werden zusätzlich zu den mittels Assumption MSDs definierten expliziten Annahmen auch implizite Annahmen über das Umgebungsverhalten getroffen, die unabhängig von der konkreten Spezifikation gelten. Die durch die Synthese erzeugte Implementierung ist nur bei Gültigkeit auch dieser Annahmen eine korrekte Implementierung der Spezifikation. Die konkreten impliziten Annahmen sind für die verschiedenen Erweiterungen des Algorithmus teilweise unterschiedlich.

Grundsätzlich wird in dieser Arbeit angenommen, dass Umgebungsereignisse nur dann auftreten können, wenn das System (gemäß der Implementierung) nichts sendet. Dies wird auch als *Synchronizitätsannahme* (engl. *synchrony hypothesis*) bezeichnet [BB91]. Die Implementierung kann also durch Senden längerer Nachrichtensequenzen die Umgebung am Reagieren hindern, muss aber immer dann auf Umgebungsnachrichten reagieren, wenn sie selbst nichts sendet. Die Übertragungsdauer von Nachrichten wird grundsätzlich als vernachlässigbar kurz angenommen.

In Abschnitt 3.4 wird beschrieben, wie architekturbedingte Einschränkungen der Kommunikation bei der Synthese berücksichtigt werden können. Diese Einschränkungen gelten auch für die Umgebung und implizieren die Annahme, dass diese keine Nachrichten sendet, die den Einschränkungen widersprechen.

Bei zeitbehafteten Systemen muss die Synchronizitätsannahme angepasst werden. Sie gilt für diese nur, wenn das System spätestens zu demselben Zeitpunkt wie die Umgebung sendet. Wenn das System jedoch länger wartet, dann erlaubt es der Umgebung, Nachrichten zu senden, die das System dann berücksichtigen muss. Ein Warten des Systems kann insbesondere dadurch erforderlich sein, dass Echtzeitanforderungen in Requirement MSDs dieses zum Warten zwingen. Im Kontext der zeitbehafteten Erweiterung des Algorithmus wird in Abschnitt 4.2.4 die *Verhinderbarkeit* als Eigenschaft von Umgebungsereignissen eingeführt. Das System muss immer dann auf Umgebungsereignisse reagieren, wenn sie nicht verhinderbar sind.

In Abschnitt 4.4 wird die zeitbehaftete Variante des Algorithmus um asynchrone Kommunikation mit Zeitbedarf der Nachrichtenübertragung erweitert. Für asynchron übermittelte Nachrichten gelten dabei abweichende Umgebungsannahmen: Der erwähnte Vorrang des Systems gilt (selbst gegenüber „verhinderbaren“ Umgebungsnachrichten) bei diesen Nachrichten nur für den Sendezeitpunkt, nicht für die Nachrichtenübermitt-

lung – während dieser kann die Umgebung also Nachrichten senden. Die minimale und maximale Dauer der Nachrichtenübermittlung und die maximal gleichzeitig in Übertragung befindlichen Nachrichten können für jede Kommunikationsverbindung als Teil der Spezifikation festgelegt werden. Diese können als eine spezielle Form von Umgebungsannahmen betrachtet werden, wenn man das Kommunikationsmedium als Teil der Umgebung betrachtet. Die Festlegung einer maximalen Anzahl in Übertragung befindlicher Nachrichten impliziert auch die Annahme, dass die Umgebung keine Nachrichten senden wird, die diese Anzahl überschreiten würden. Weiterhin wird angenommen, dass alle Nachrichten über dieselbe Verbindung in derselben Reihenfolge ankommen, in der sie gesendet werden (sich also nicht gegenseitig „überholen“).

2.2 Controllersysteme als Implementierungsmodelle

Während sich der vorherige Abschnitt mit MSD-Spezifikationen als der Eingabe des Syntheseverfahrens befasste, behandelt dieser Abschnitt die Ausgabe des Verfahrens im Erfolgsfall. Diese ist ein *Controllersystem*, ein Netzwerk kommunizierender Controller, das die Anforderungen an eine korrekte Implementierung der MSD-Spezifikation aus Abschnitt 2.1.6 erfüllt. Zunächst werden in Abschnitt 2.2.1 Controllersysteme und Controller allgemein definiert. Anschließend wird in Abschnitt 2.2.2 eine Variante von Controllern auf Basis von Timed Automata eingeführt, mittels der auch zeitbehaftetes Verhalten definiert werden kann. Zudem unterstützt diese Variante neben synchroner auch asynchrone Kommunikation. Schließlich behandelt Abschnitt 2.2.3 die Frage, wann ein Controllersystem eine korrekte Implementierung einer MSD-Spezifikation ist.

2.2.1 Controllersysteme und Controller

Ein Controller ist ein deterministischer endlicher Automat (DFA, engl. *deterministic finite automaton*), der für die von ihm kontrollierten Subsysteme festlegt, welche Nachrichtensequenzen diese als Reaktion auf empfangene Nachrichtensequenzen senden. Die Transitionen dieses Automaten sind mit den gesendeten bzw. empfangenen Nachrichtenereignissen beschriftet. Diese Nachrichtenereignisse entsprechen den in Abschnitt 2.1.2 für MSD-Spezifikationen definierten, d. h. sie definieren jeweils Name, Sender und Empfänger der übermittelten Nachricht, bei parametrisierten Nachrichten zusätzlich die Werte der Parameter. Zur Abgrenzung gegenüber den Zuständen und Transitionen anderer Zustandsgraphen werden die Zustände und Transitionen eines Controllers in dieser Arbeit auch als Controllerzustände und Controllertransitionen bezeichnet.

Definition 1 (Controllersysteme)**.** Ein *Controllersystem* besteht aus einer Menge von Subsystemen, die jeweils kontrollierbar oder unkontrollierbar sein können, und einer Menge von *Controllern*, die das Verhalten der kontrollierbaren Subsysteme definieren. Dabei ist jedem Controller eine (nicht-leere) Menge von kontrollierbaren Subsystemen zugeordnet, die er *kontrolliert*. Jedes kontrollierbare Subsystem muss von genau einem Controller des Controllersystems kontrolliert werden.

In Abschnitt 2.1.2 wurden die Begriffe „kontrollierbar“ und „beobachtbar“ für Nachrichten eingeführt, die das System senden bzw. senden oder empfangen kann. Diese dort

auf das gesamte System bezogenen Eigenschaften sind für die verteilte Synthese auch auf Ebene der einzelnen Controller und für deren Transitionen relevant. Daher werden Beobachtbarkeit und Kontrollierbarkeit nachfolgend auch für diese definiert.

Definition 2 (Kontrollierbarkeit und Beobachtbarkeit)**.** Eine Nachricht wird als *kontrollierbar durch ein Subsystem* bezeichnet, wenn dieses Subsystem die Nachricht senden kann. Eine Nachricht wird als *beobachtbar für ein Subsystem* bezeichnet, wenn das Subsystem die Nachricht entweder senden oder empfangen kann. Eine Nachricht ist *kontrollierbar bzw. beobachtbar für einen Controller*, wenn diese Nachricht für mindestens ein durch den Controller kontrolliertes Subsystem kontrollierbar bzw. beobachtbar ist. Ein Nachrichtenereignis ist *kontrollierbar/beobachtbar*, wenn die dazugehörige Nachricht kontrollierbar/beobachtbar ist. Eine Transition ist *kontrollierbar* bzw. *beobachtbar*, wenn das Ereignis der Transition für den Controller der Transition kontrollierbar bzw. beobachtbar ist.

Die für einen Controller unkontrollierbaren beobachtbaren Ereignisse sind seine Eingaben, die für ihn kontrollierbaren Ereignisse sind seine Ausgaben. Da jeder Controller ein DFA ist, darf es in jedem Zustand eines Controllers maximal eine ausgehende kontrollierbare Transition geben.

Die graphische Repräsentation eines Controllers ist, wie für DFA üblich, ein gerichteter Graph, in dem Zustände durch Knoten und Transitionen durch Kanten dargestellt werden. Der Startzustand (hier Zustand `0`) ist gesondert hervorgehoben. Die beim Schalten einer Transition gesendeten bzw. empfangenen Ereignisse werden an der Transition annotiert. Dabei wird das Senden mit dem Suffix `!` und das Empfangen mit `?` gekennzeichnet.

Abbildung 2.9 zeigt einen beispielhaften Controller für die Presse aus dem in Abschnitt 1.1 eingeführten Beispiel des Produktionssystems. Der gezeigte Controller empfängt im Startzustand eine Nachricht `blankAtPress` und sendet als Reaktion darauf erst `press` und dann `allowPlateToBelt`. In der graphischen Repräsentation wird auf die Angabe der Sender und Empfänger der Ereignisse verzichtet, solange diese durch den Nachrichtennamen in diesem Kontext bereits feststehen – also für diesen Nachrichtennamen in allen MSDs gleich sind.

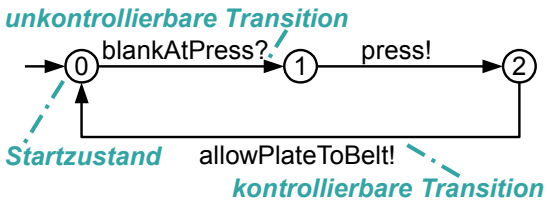


Abbildung 2.9: Beispiel für einen Controller

Die Semantik eines Controllers ist wie folgt definiert: Tritt im System ein Ereignis auf, das zu einer unkontrollierbaren ausgehenden Transition des aktuellen Controllerzustands gehört, dann schaltet diese Transition sofort. Wird das Ereignis also von der Umgebung gesendet, dann legt diese den Zeitpunkt des Schaltens fest, andernfalls der Controller des

sendenden Subsystems. Beim Schalten einer Transition wechselt der Controller in den Folgezustand der Transition. Beim Schalten einer unkontrollierbaren Transition wird zudem das Ereignis konsumiert, welches das Schalten ausgelöst hat.

Existiert im aktuellen Zustand eines Controllers eine kontrollierbare ausgehende Transition, so schaltet diese irgendwann. Beim Schalten einer kontrollierbaren Transition wird das ihr zugeordnete Ereignis ausgelöst. Wird dieses Ereignis von einem Subsystem empfangen, das durch einen anderen Controller kontrolliert wird, so führt dies – wie oben beschrieben – sofort zum Schalten der empfangenden Transition dieses Controllers. Wird das Ereignis von einem Subsystem empfangen, das vom demselben Controller wie der Sender des Ereignisses kontrolliert wird, dann wird das Ereignis sofort konsumiert. Auch wenn im System ein Ereignis auftritt, zu dem keine unkontrollierbare ausgehende Transition in irgendeinem Controller existiert, dann wird dieses sofort konsumiert. Dies gilt insbesondere für Ereignisse, deren Empfänger die Umgebung ist.

Wie bei MSD-Spezifikationen (vgl. Abschnitt 2.1.2) wird auch für Controller ein Kommunikationsmodell angenommen, in dem Nachrichten synchron übermittelt werden. Senden und Empfangen werden also als ein einziges Ereignis betrachtet, welches zudem keine Zeit benötigt. Analog zu MSD-Spezifikationen wird angenommen, dass die Umgebung keine Nachrichten sendet, solange sich irgendein Controller in einem Zustand mit einer kontrollierbaren ausgehenden Transition befindet (siehe Abschnitt 2.1.6). Gelten diese Annahmen nicht, dann sind die Controller möglicherweise keine korrekte Implementierung der MSD-Spezifikation, selbst wenn sie als Ergebnis der Synthese erzeugt wurden.

2.2.2 Zeitbehaftete Controller auf Basis der Timed Automata

Um Implementierungen für MSD-Spezifikationen zu beschreiben, die auch Timed MSDs (siehe Abschnitt 2.1.5) enthalten, wird hier das Konzept der Controller um Modellelemente zur Definition von Echtzeitverhalten erweitert. Als Basis wird dabei das bekannte Modell der *Timed Automata* genommen, die nachfolgend zuerst beschrieben werden. Anschließend werden *Clock Zones* vorgestellt, eine symbolische Repräsentation von Zeit, über die die Semantik von *Netzwerken von Timed Automata* definiert ist. In dieser Arbeit werden sie zusätzlich zur Modellierung des Verhaltens von zeitbehafteten MSD-Spezifikationen eingesetzt. Abschließend werden die geringfügigen Unterschiede zwischen *zeitbehafteten Controllern* und gewöhnlichen Timed Automata diskutiert.

Timed Automata

Um zeitbehaftetes zustandsbasiertes Verhalten formal beschreiben zu können, wurden *Timed Automata* von Alur und Dill [AD94] als Erweiterung der endlichen Automaten eingeführt. Timed Automata sind endliche Automaten, die zusätzlich *Uhrenvariablen* (oder einfach *Uhren*) und Modellelemente, welche sich auf Uhren beziehen, beinhalten können.

Uhrenvariablen in Timed Automata sind ähnlich definiert wie in Timed MSDs, die bereits in Abschnitt 2.1.5 beschrieben wurden: Die initial mit 0 belegten Variablen steigen kontinuierlich und mit konstanter Rate (die für alle Uhren gleich ist) im Wert

und können mittels eines *Clock Reset* auf 0 zurückgesetzt werden. Anders als in Timed MSDs existieren Uhren in Timed Automata jedoch bereits ab dem Systemstart. Sie modellieren daher entweder die seit Systemstart vergangene Zeit, oder den Zeitabstand seit dem letzten Clock Reset. In Timed Automata beziehen sich Clock Resets immer auf Transitionen. Sie werden in der Form `{c1, c2, ..., cn}` für Uhren $c_1, c_2, \ldots, c_n$ an der betreffenden Transition annotiert. Beim Schalten dieser Transition werden dann alle Uhren der definierten Menge auf 0 zurückgesetzt.

Außer Clock Resets können Transitionen in Timed Automata einen *Time Guard* definieren. Dieser hat die Form `[cond]`, wobei `cond` eine Konjunktion von Uhrenvergleichen ist, also $cond = cmp_1 \wedge cmp_2 \wedge \ldots cmp_n$ für Uhrenvergleiche $cmp_1, cmp_2, \ldots, cmp_n$. Uhrenvergleiche haben (wie in Timed MSDs) den Aufbau `<Uhr> <Operator> <Uhrwert>` für eine Uhr `<Uhr>`, einen Vergleichsoperator `<Operator>` $\in \{<, \leq, =, \geq, >\}$ und einen Vergleichswert `<Uhrwert>` $\in \mathbb{N}$.

Time Guards schränken das Schalten einer Transition auf die Zeitpunkte ein, zu denen die Bedingung `cond` erfüllt ist. Dies bedeutet, dass alle in der Bedingung referenzierten Uhren einen Wert haben müssen, für den alle auf diese Uhr bezogenen Vergleiche in `cond` erfüllt sind. Anders als Zeitbedingungen in Timed MSDs definieren Time Guards in Timed Automata also keine Anforderungen an ein Verhalten, sondern sie schränken das durch den Automaten definierte Verhalten direkt ein.

Die Modellelemente der Timed Automata, die den Zuständen der endlichen Automaten entsprechen, werden als *Locations* bezeichnet. Um zu modellieren, dass ein Timed Automaton eine Location bis zu einem bestimmten Zeitpunkt verlassen muss, können *Invarianten* an dieser Location definiert werden. Sie werden im Rahmen dieser Arbeit als Beschriftung der Form `I: cond` an der jeweiligen Location annotiert. Wie bei Time Guards ist `cond` eine Konjunktion von Uhrenvergleichen. Bei Invarianten sind jedoch die möglichen Vergleichsoperatoren auf `<Operator>` $\in \{<, \leq, \}$ eingeschränkt.

Eine Invariante an einer Location modelliert, dass der Automat diese Location in jedem Fall verlässt, bevor die Bedingung der Invariante verletzt wird. Ohne das Schalten einer ausgehenden Transition wird das zwangsläufig irgendwann der Fall sein, da Invarianten obere Schranken für Uhrenvariablen definieren und diese unbegrenzt im Wert steigen, wenn sie nicht durch das Schalten einer Transition mit einem Clock Reset zurückgesetzt werden.

Um eine Verletzung der Invarianten einer Location zu vermeiden, muss ein Timed Automaton also eine ausgehende Transition der Location schalten, bevor der Time Guard der letzten schaltbaren Transition nicht mehr erfüllt ist. Verweilt ein Timed Automaton länger in der Location, so tritt bei Erreichen der durch die Invariante definierten oberen Schranke ein *Time Stopping Deadlock* auf: In diesem kann einerseits die Location nicht mehr verlassen werden, aber andererseits auch keine Zeit mehr im System vergehen, weil sonst die Invariante verletzt würde. Werden Timed Automata als Implementierung verwendet, so müssen diese so konstruiert sein, dass dieser Fall nicht auftritt.

Es kann erforderlich sein, dass eine Location eines Timed Automaton sofort nach ihrem Betreten wieder verlassen wird. Hierzu kann eine zusätzliche Uhr `c` eingeführt werden, die an allen eingehenden Transitionen der Location zurückgesetzt wird. Eine Invariante `c` $\leq$ `0` der Location stellt dann sicher, dass in dieser keine Zeit vergehen kann und ein sofortiges Schalten einer ausgehenden Transition erfolgen muss. Derselbe Effekt

kann durch Deklarieren der Location als *urgent* erzielt werden [BDL04]. Dies wird im Rahmen dieser Arbeit durch ein Label u an der Location gekennzeichnet.

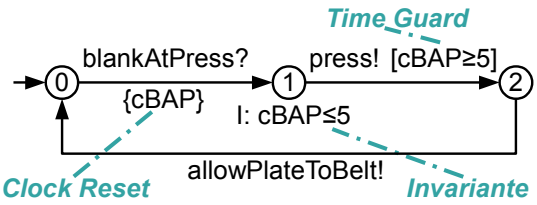


Abbildung 2.10: Beispiel für einen Timed Automaton bzw. zeitbehafteten Controller

Abbildung 2.10 zeigt die graphische Repräsentation eines Timed Automaton, der die Presse aus dem Beispiel des Produktionssystems kontrolliert. Es handelt sich um die zeitbehaftete Variante des Controllers aus Abbildung 2.9. Hier wird die Uhr `cBaP` beim Empfangen von `blankAtPress` durch einen Clock Reset auf 0 zurückgesetzt. Die Invariante in Zustand `1` und der Time Guard an der ausgehenden Transition definieren hier, dass nach genau 5 Sekunden `press` gesendet wird.

Time Guards und Invarianten können aus Sicht einer konkreteren Implementierung des Timed Automaton (z. B. in Form von Binärcode) als Anforderungen betrachtet werden: Schaltet die Implementierung eine Transition beispielsweise außerhalb des Zeitraums, in dem ihr Time Guard gültig ist, so handelt es sich nicht um eine korrekte Implementierung. Ebenso ist es Aufgabe der Implementierung, ein rechtzeitiges Verlassen jedes Zustands vor Ablauf der Invarianten zu garantieren.

Timed Automata können über *Synchronisationskanäle* miteinander kommunizieren. Durch eine Beschriftung an einer Transition kann für diese eine *Synchronisation* definiert werden: Mittels der Notation `s!` kann über einen Synchronisationskanal mit dem Namen `s` gesendet, mit `s?` über diesen empfangen werden. In dieser Arbeit wird angenommen, dass für jede Transition maximal eine Synchronisation definiert werden kann.

Transitionen mit Synchronisationen können nur paarweise schalten. Dabei muss eine Transition auf demselben Synchronisationskanal senden, auf dem die andere empfängt. Es muss also für denselben Kanal `s` die eine Transition mit `s!` und die andere mit `s?` beschriftet sein. Gegebenenfalls vorhandene Time Guards der Transitionen müssen gleichzeitig erfüllt sein. Beide Transitionen schalten dann – wenn sie schalten – synchron, also gleichzeitig. Sie müssen jedoch nicht schalten, sofern durch den Verzicht auf das Schalten keine Invariante verletzt wird.

Eine parallele Komposition von Timed Automata, die miteinander per Synchronisation kommunizieren können, wird als *Netzwerk von Timed Automata* bezeichnet. Ein solches hat aufgrund der reellwertigen Uhren in Timed Automata einen unendlich großen Zustandsraum. Daher wird die Semantik von Netzwerken von Timed Automata mittels *Clock Zones* definiert, welche die Zeit im System symbolisch repräsentieren. Diese werden im nächsten Abschnitt näher erläutert.

Die möglichen Zustände eines Netzwerks von Timed Automata können mittels eines *Zone-Graphen* repräsentiert werden [BY04]. Jeder Zustand des Zone-Graphen wird durch eine Kombination der aktuellen Locations aller Timed Automata und einer Clock

Zone charakterisiert. Diese Clock Zone repräsentiert die möglichen Wertebereiche aller Uhren der Timed Automata im betreffenden Zustand. Im initialen Zustand des Zone-Graphen sind alle Timed Automata in ihren initialen Locations und alle Uhren sind 0. Weitere Zustände sind über drei Arten von Transitionen erreichbar, die das Vergehen von Zeit im System, das Schalten einzelner Transitionen und das gleichzeitige Schalten von zwei Transitionen in einer Synchronisation modellieren.

Clock Zones

Eine *Clock Zone* [BY04] modelliert die möglichen Belegungen aller Uhrenvariablen in einem Zustand eines Systems. Dazu beschränkt sie den Wertebereich (Minimal- und Maximalwert) der Uhren selbst und den Wertebereich der Differenz zwischen den Uhren (d. h. wie viel höher/niedriger der Wert einer Uhr als der einer anderen Uhr sein darf).

Eine Clock Zone definiert Bedingungen für eine Menge von Uhren, die auch als *Clock Constraints* bezeichnet werden. Diese Bedingungen werden als Konjunktion von Ungleichungen über die Uhrenwerte ausgedrückt. Diese Ungleichungen haben entweder die Form $c \odot x$ oder die Form $c - c1 \odot x$ für Uhren c und $c1$, einen Vergleichsoperator $\odot \in \{<,\ \leq,\ =,\ \geq,\ >\}$ und einen Vergleichswert $x \in \mathbb{N}$. Die Konjunktion ist genau dann erfüllt, wenn sich alle Uhrenwerte innerhalb der Clock Zone befinden. In der Clock Zone `true` liegen also beliebige Uhrenwerte, während in der Clock Zone `false` überhaupt keine Uhrenwerte enthalten sind. Eine einzelne Clock Zone ist immer konvex, also ein zusammenhängender Zeitbereich. Um auch unzusammenhängende Zeitbereiche zu definieren, wird daher eine Disjunktion von Clock Zones benötigt. Eine solche wird im Rahmen dieser Arbeit als *Federation* [EH11] bezeichnet.

Zur Manipulation von Clock Zones bzw. Federations können verschiedene ein- und zweistellige Operationen verwendet werden. Diese werden hier kurz zusammengefasst; für eine detailliertere Darstellung wird auf Bengtsson und Yi [BY04] verwiesen. Diese Operationen und die hier verwendeten Operanden werden für die zeitbehaftete Erweiterung des Syntheseverfahrens in Kapitel 4 benötigt. Die Operationen sind auf Clock Zones, Federations und einfachen Clock Constraints definiert, und liefern als Ergebnis im Allgemeinen eine Federation (die aber im Speziellen auch nur eine einzelne Clock Zone enthalten kann). Nachfolgend wird die Anwendung der Operationen auf Federations f, f' diskutiert.

Die Operation $f^{\uparrow}$ liefert eine Federation, aus der die oberen Schranken, also alle Bedingungen $\{<, \leq\}$ (und der $\leq$-Anteil von $=$) entfernt sind und modelliert damit das Verstreichen von beliebig viel Zeit. Analog entsteht die als Ergebnis von $f^{\downarrow}$ zurückgegebene Federation durch Entfernen aller unteren Schranken (also $\{>, \geq\}$, unter Beibehaltung der Einschränkung $\geq$ `0` für alle Uhren. Darin sind alle Uhrenwerte enthalten, von denen aus f durch Verstreichen von Zeit erreichbar ist. Durch die Operation $f \wedge f'$ (Konjunktion, „schneiden“ von Federations) entsteht eine Federation, die alle Uhrenwerte enthält, die sowohl in f als auch in f' enthalten sind. Die Operation $f \vee f'$ (Disjunktion) liefert als Ergebnis eine Federation, welche alle Uhrenwerte enthält, die in f oder f' enthalten sind. Überlappende Clock Zones aus f und f' können dadurch auch zu einer einzigen Clock Zone „verschmelzen“. Die Operation $\neg f$ liefert die Negation von f, welche alle Uhrenwerte enthält, die nicht in f enthalten sind. Die Operation $[r \leftarrow 0]f$ erzeugt für

eine Uhrenmenge r eine Federation, in der alle Uhren in r auf den Wert 0 beschränkt sind. Diese Operation modelliert damit einen Clock Reset dieser Uhren. Die Operation $f - f'$ (Subtraktion) schließlich ist identisch zu $f \wedge \neg f'$, enthält also alle Uhrenwerte in f, die nicht in f' sind.

Zeitbehaftete Controller

Ein *zeitbehafteter Controller* ist ein Timed Automaton, der die Rolle eines Controllers in einem Controllersystem einnimmt. Synchronisationen entsprechen dabei dem Senden (`!`) bzw. Empfangen (`?`) einer Nachricht mit dem Namen des Synchronisationskanals. Weiterhin wird das Konzept der kontrollierbaren bzw. unkontrollierbaren Transitionen der Controller nach der Definition aus Abschnitt 2.2.1 übernommen. Im Folgenden werden zusätzliche Eigenschaften aufgeführt, die ein Timed Automaton erfüllen muss, damit dieser ein zeitbehafteter Controller ist. Die Locations von zeitbehafteten Controllern werden hier analog zu den nicht zeitbehafteten Controllern als Zustände bezeichnet. Die graphische Repräsentation eines zeitbehafteten Controllers entspricht derjenigen eines gewöhnlichen Timed Automaton (siehe Abbildung 2.10).

In MSD-Spezifikationen wird vorausgesetzt, dass ein Subsystem, das Empfänger einer Nachricht ist, das Empfangen dieser Nachricht nicht verhindern kann. Da Empfangen und Senden ein einziges Ereignis sind, würde durch ein Blockieren des Empfangens auch das Senden verhindert. Wäre dies möglich, so könnten Controller durch Verhindern der minimalen Nachricht eines MSDs dieses MSD auf triviale Weise erfüllen. Um die Semantik der MSDs korrekt umzusetzen, müssen zeitbehaftete Controller daher immer dann eine Transition mit empfangender Synchronisation schalten können, wenn ein Sender über diesen Synchronisationskanal senden kann. Der zeitbehaftete Controller muss zu diesen Nachrichten dann also jeweils eine ausgehende Transition im aktuellen Zustand haben, die eine `?`-Synchronisation für die jeweilige Nachricht definiert und deren Time Guard erfüllt ist. In graphischen Repräsentationen von zeitbehafteten Controllern in dieser Arbeit gilt: Ist eine solche Transition nicht explizit eingezeichnet, dann gilt eine entsprechende Selbsttransition als definiert.

Bei nicht zeitbehafteten Controllern wird angenommen, dass diese irgendwann senden, wenn sie senden können (siehe Abschnitt 2.2.1). Dies ist der Fall, wenn im aktuellen Zustand eine kontrollierbare ausgehende Transition existiert. Timed Automata können jedoch beliebig lange in einer Location verweilen, wenn dadurch keine Invariante verletzt wird. Um zu gewährleisten, dass ein Timed Automaton eine ausgeführte Nachricht irgendwann sendet, reicht es also nicht aus, wenn dieser in der betreffenden Location eine passende Transition zum Senden der Nachricht definiert. Er muss zusätzlich durch eine Invariante zum Verlassen der Location und damit zum Senden gezwungen werden.

Um MSD-Spezifikationen implementieren zu können, müssen zeitbehaftete Controller deshalb in allen Zuständen mit kontrollierbaren ausgehenden Transitionen Invarianten enthalten. Ist dies bei zeitbehafteten Controllern in dieser Arbeit nicht durch explizit definierte Invarianten der Fall, so gilt im betreffenden Zustand eine spezielle Invariante als automatisch definiert: Diese Invariante enthält eine beliebig hohe, aber feste Schranke, die irgendwann erreicht wird und ein Schalten erzwingt. Die Invariante bezieht sich dabei auf eine spezielle Uhr, die beim Betreten des Zustands zurückgesetzt wird. Durch

diese Modellierung wird sichergestellt, dass auch zeitbehaftete Controller irgendwann senden müssen, wenn sie senden können – ohne eine Semantik annehmen zu müssen, die von derjenigen der Timed Automata abweicht.

Für zeitbehaftete Systeme werden in dieser Arbeit neben synchronen auch asynchrone Nachrichten betrachtet, die Zeit benötigen können und die durch jeweils zwei Ereignisse repräsentiert werden (siehe Abschnitt 4.4). Wird eine solche Nachricht durch Controller gesendet bzw. empfangen, dann schaltet die sendende Transition beim Auftreten des Sendeereignisses, die empfangende beim Auftreten des Empfangsereignisses. In dieser Arbeit wird in der graphischen Repräsentation von Controllern auch bei asynchronen Ereignissen `?` bzw. `!` verwendet, um das Empfangen bzw. Senden des Ereignisses kenntlich zu machen.

2.2.3 Controllersysteme als Implementierungen von MSD-Spezifikationen

Ein Controllersystem ist dann eine korrekte Implementierung einer MSD-Spezifikation, wenn es die Kriterien aus Abschnitt 2.1.6 für eine solche erfüllt. Alle Abläufe, die für das Controllersystem in Ausführung mit seiner Umgebung möglich sind, müssen also die Spezifikation erfüllen. Die Abläufe eines nicht zeitbehafteten Controllers sind durch die Nachrichtensequenzen definiert, die bei seiner Ausführung mit der Umgebung auftreten können. Die Abläufe eines zeitbehafteten Controllers beinhalten zudem die zeitlichen Abstände zwischen den Nachrichten.

Abschnitt 3.2.2 definiert genauere Kriterien für die korrekte Implementierung einer MSD-Spezifikation durch ein Controllersystem. Diese Kriterien beziehen sich dabei auf ein vom Synthesealgorithmus verwendetes Modell des globalen Zustandsraums einer MSD-Spezifikation. In Abschnitt 4.3.1 wird diese Definition auf zeitbehaftete Controllersysteme und zeitbehaftete MSD-Spezifikationen erweitert.

Kapitel 3

Synthese verteilter Systeme

Dieses Kapitel stellt ein Verfahren zur automatischen Synthese von Implementierungsmodellen verteilter Systeme aus szenariobasierten Anforderungsspezifikationen vor. Als Eingabe wird dabei eine MSD-Spezifikation vorausgesetzt, also eine Spezifikation auf Basis von Modal Sequence Diagrams (MSDs), wie sie in Abschnitt 2.1 eingeführt wurden. Sofern eine verteilte Realisierung der Spezifikation möglich ist, so wird als Ausgabe des Verfahrens für jedes Subsystem ein Automat erzeugt, der dessen Verhalten beschreibt. Das gesamte Systemverhalten, bestehend aus allen erzeugten Automaten und dem durch Annahmen modellierten Verhalten der Umgebung, erfüllt dann die gegebene Spezifikation. Ist keine verteilte Realisierung möglich, so wird dies dem Benutzer angezeigt. Der Hauptvorteil des Verfahrens gegenüber den in Kapitel 7 behandelten verwandten Arbeiten ist, dass bei Bedarf automatisch zusätzliches Kommunikationsverhalten erzeugt wird, welches in der Spezifikation nicht explizit definiert wurde, aber dennoch zu ihrer Umsetzung erforderlich ist.

Der Inhalt dieses Kapitels wurde in Teilen bereits in einem Konferenzbeitrag veröffentlicht [BGS15], der im Rahmen dieser Arbeit entstanden ist. Gegenüber dieser Veröffentlichung ist die Beschreibung hier deutlich ausführlicher. Insbesondere werden Spezialfälle genauer betrachtet.

Dieses Kapitel beschreibt zunächst eine Basisvariante des Syntheseverfahrens, die keine Echtzeitanforderungen unterstützt. Zudem wird hier ausschließlich synchrone Kommunikation angenommen, d. h. das Senden einer Nachricht wird zusammen mit ihrem Empfangen als einzelnes atomares Ereignis betrachtet. In Kapitel 4 wird das Verfahren dann um die Berücksichtigung von Echtzeitanforderungen und asynchroner Kommunikation erweitert.

Das Kapitel ist wie folgt unterteilt: Zunächst wird in Abschnitt 3.1 eine MSD-Spezifikation für das in Abschnitt 1.1 eingeführte Anwendungsbeispiel einer Produktionszelle vorgestellt, anhand der das Verfahren erläutert wird. Abschnitt 3.2 gibt einen Überblick über einerseits das Problem der verteilten Synthese und andererseits über den zur Lösung des Problems entwickelten Algorithmus und grenzt diesen von Vorarbeiten ab. Auf dieser Grundlage wird dann in Abschnitt 3.3 der Synthesealgorithmus im Detail vorgestellt und anhand einer Beispielausführung für das Anwendungsbeispiel erläutert. Dabei wird zunächst von einer Architektur ausgegangen, in der die Kommunikation zwischen den Subsystemen nicht eingeschränkt ist. Abschnitt 3.4 erweitert den Syntheseansatz so, dass dieser architekturbedingte Einschränkungen der Kommunikation zwischen Subsystemen berücksichtigen kann. Schließlich fasst Abschnitt 3.5 das Kapitel zusammen.

3.1 Spezifikation des Anwendungsbeispiels

Um die Ausführung der Synthese anhand eines Beispiels zeigen zu können, wird hier eine beispielhafte MSD-Spezifikation vorgestellt, zu der durch den vorgestellten Algorithmus eine verteilte Implementierung erzeugt werden soll. Diese Spezifikation formalisiert die Systemanforderungen und Umgebungsannahmen des in Abschnitt 1.1 eingeführten Anwendungsbeispiels einer Produktionszelle in einer einfachen Variante.

Abbildung 3.1 zeigt (unten) eine Skizze der Produktionszelle: Metallrohlinge kommen auf einem Zuführförderband an und werden von einem Sensor als intakt oder defekt erkannt. Ein Roboterarm sortiert defekte Rohlinge aus und bewegt die intakten zu einer Presse, die diese presst und damit in fertige Metallplatten umwandelt. Diese Metallplatten werden vom Roboterarm an der Presse aufgenommen und auf einem Ablageförderband abgelegt. Neue Rohlinge kommen immer erst dann beim Sensor am Zuführförderband an, wenn der vorherige Rohling bereits bearbeitet wurde.

Die zur Umsetzung des beschriebenen Verhaltens zwischen den Subsystemen ausgetauschten Nachrichten sind in der Skizze eingezeichnet. Diese Nachrichten sind an Pfeilen annotiert, die die Kommunikationsrichtung angeben. Die Anforderungen an dieses Kommunikationsverhalten und die Annahmen an die Umgebung werden sowohl textuell beschrieben, als auch als MSD-Spezifikation (siehe Abschnitt 2.1) formalisiert.

Zunächst wird dazu in Abschnitt 3.1.1 die Struktur des Beispiels formalisiert und die kontrollierbaren und unkontrollierbaren Subsysteme werden in Form der Systemobjekte und Umgebungsobjekte definiert. Für die Systemobjekte werden dann in Abschnitt 3.1.2 mittels Requirement MSDs Anforderungen definiert, während für die Umgebungsobjekte in Abschnitt 3.1.3 mittels Assumption MSDs Annahmen definiert werden. Zur Implementierung dieser Spezifikation ist zusätzliche (nicht explizit spezifizierte) Kommunikation erforderlich. Daher wird abschließend in Abschnitt 3.1.4 diskutiert, weshalb dies der Fall ist und an welcher Stelle diese Kommunikation ergänzt werden muss.

3.1.1 Struktur

Abbildung 3.1 zeigt die Struktur der Produktionszelle oben in Form eines UML-Klassendiagramms auf Typebene und mittig in Form eines UML-Kompositionsstrukturdiagramms (CSD) auf Instanzebene. Die Bedeutung der Diagramme und der darin definierten Elemente wurde in Abschnitt 2.1.1 erläutert.

Das Beispiel enthält nur zwei Subsysteme, die durch die Software direkt gesteuert werden können: Dies sind die Steuergeräte für den Roboterarm (ArmController aC) und für die Presse (PressController pC). Diese sind daher als Systemobjekte definiert. Aufgabe der Synthese ist es, für diese beiden Objekte jeweils einen Controller zu erzeugen. Die Umgebungsobjekte Presse (Press p) und Arm (Arm a) sind Aktoren, denen durch Nachrichten Befehle erteilt werden können. Die von einem Objekt empfangbaren Nachrichten entsprechen den für seine Klasse definierten Operationen. Der Sensor (Sensor s) andererseits kann keine Nachrichten empfangen, sendet aber selbst Nachrichten. Die Förderbänder und die transportierten Rohlinge und Metallplatten sind ebenfalls Bestandteile der Umgebung; sie treten hier aber nicht direkt mit den Systemobjekten in Interaktion und wurden deshalb im CSD und im Klassendiagramm nicht modelliert.

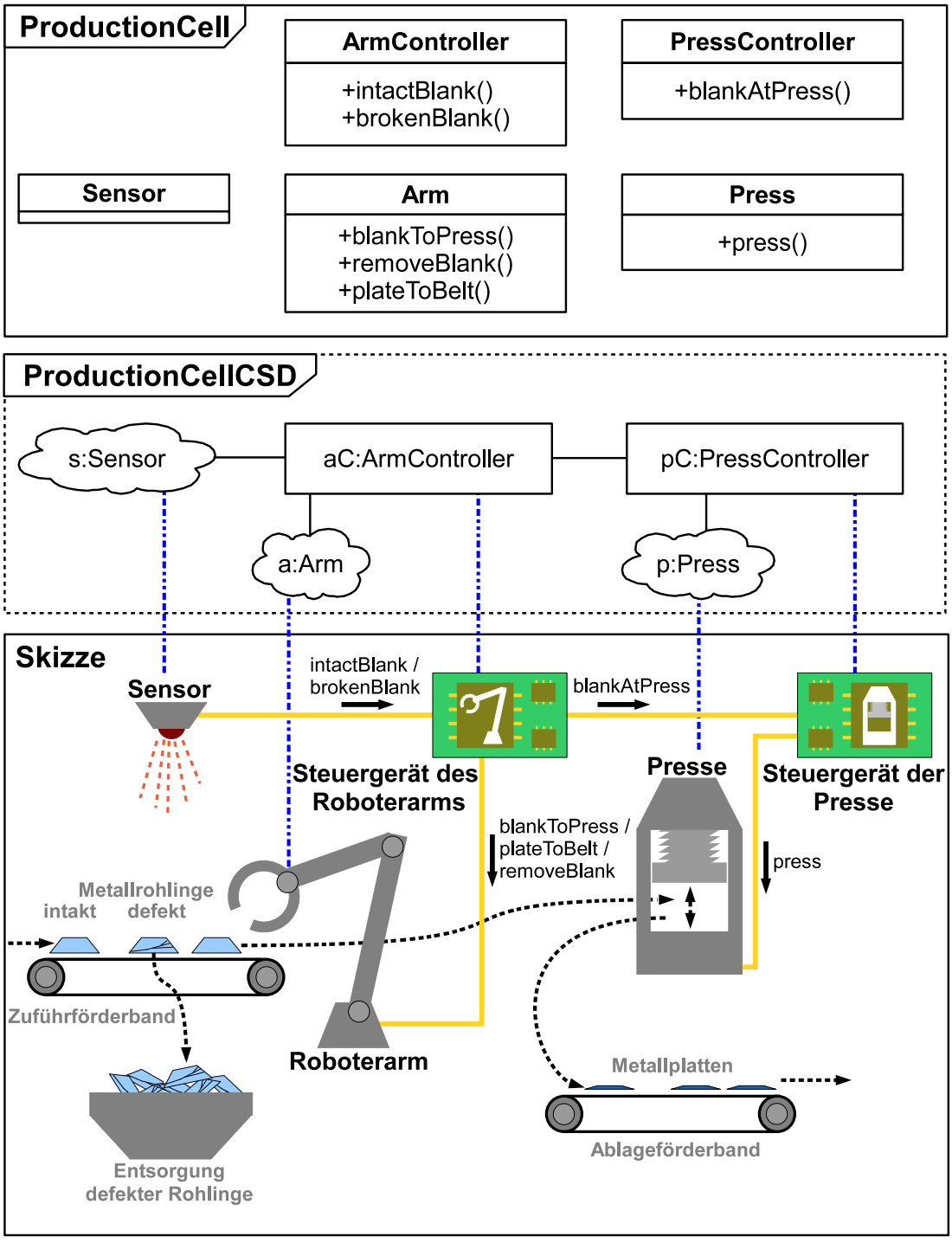
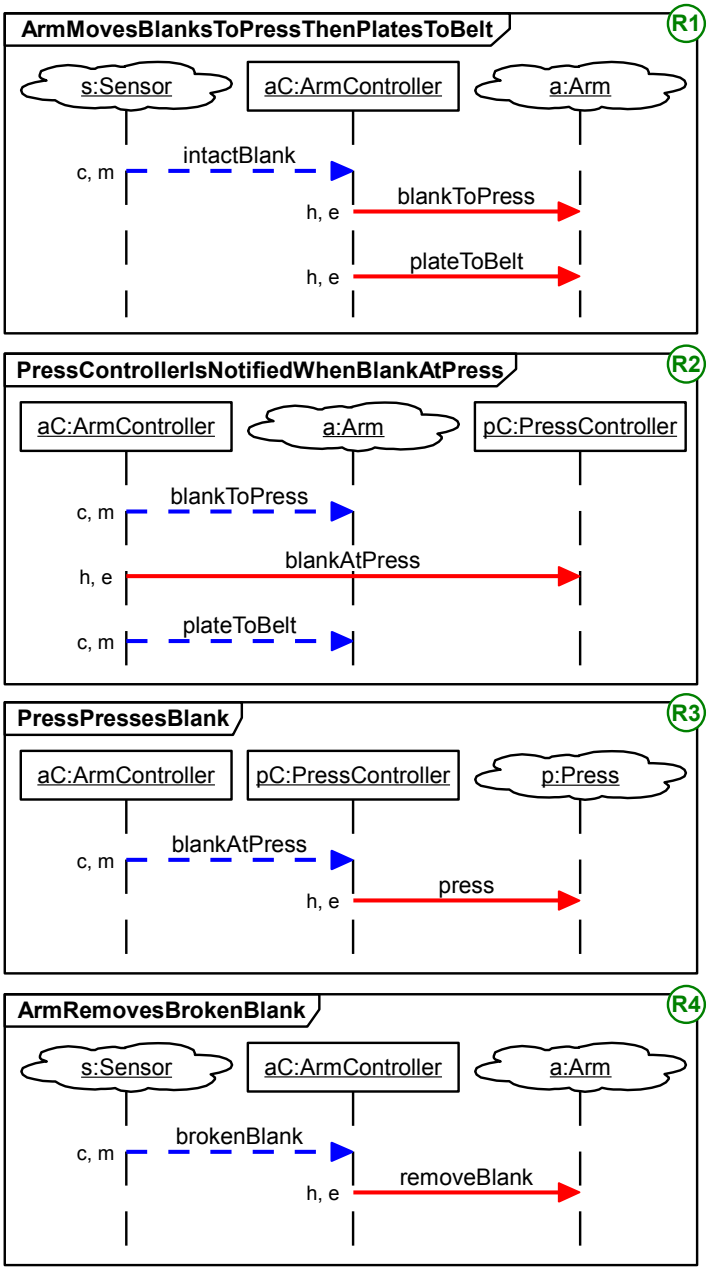


Abbildung 3.1: Struktur des Systems auf Typebene (oben), Instanzebene (Mitte) und als Skizze (unten)



„Wenn der Sensor am Zuführförderband einen neuen intakten Rohling meldet, dann muss das Steuergerät des Roboterarms den Arm anweisen, zunächst den Rohling zur Presse und dann die fertige Metallplatte zum Ablageförderband zu bewegen.“

„Wenn das Steuergerät des Roboterarms den Arm anweist, einen Rohling zur Presse zu bewegen, dann muss es anschließend – und bevor es die nächste Anweisung an den Arm sendet – das Steuergerät der Presse darüber informieren, dass sich der Rohling nun an der Presse befindet.“

„Wenn das Steuergerät der Presse durch das Steuergerät des Roboterarms darüber informiert wird, dass sich ein Rohling an der Presse befindet, dann muss das Steuergerät der Presse diese zum Pressen veranlassen.“

„Wenn der Sensor am Zuführförderband einen neuen defekten Rohling meldet, dann muss das Steuergerät des Roboterarms den Arm anweisen, diesen Rohling auszusortieren.“

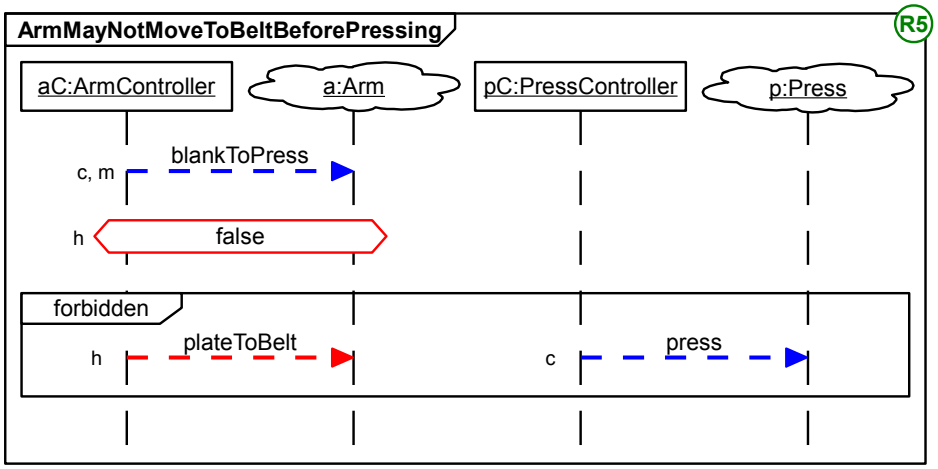
Abbildung 3.2: MSDs für die Anforderungen **R1** bis **R4** an die Produktionszelle

3.1.2 Anforderungen

Abbildung 3.2 zeigt auf der linken Seite die Requirement MSDs, welche die Anforderungen **R1** bis **R4** an das Kommunikationsverhalten der Produktionszelle modellieren. Auf der rechten Seite sind dieselben Anforderungen jeweils informell in natürlichsprachlichem Text beschrieben. Abbildung 3.3 zeigt das MSD und die natürlichsprachliche Entsprechung zu Anforderung **R5**.

Die Anforderungen **R1** und **R4** modellieren jeweils, dass der ArmController aC als Reaktion auf eine Nachricht von Sensor s bestimmte Nachrichten an den Arm a senden muss. Die Anforderung **R2** verlangt, dass aC zwischen den beiden in **R1** geforderten Nachrichten eine Nachricht an PressController pC senden muss. Gemäß **R3** muss dieser darauf mit einer Nachricht an Press p reagieren. Die in den MSDs „verlangten“, also ausgeführten Nachrichten sind außerdem jeweils heiß, müssen also in der definierten Reihenfolge auftreten.

Die etwas komplexere Anforderung **R5** in Abbildung 3.3 modelliert ein Verbot für den ArmController aC, die Nachricht `plateToBelt` zu senden, was durch eine heiße verbotene Nachricht modelliert wird. Dieses Verbot gilt ab dem Senden von `blankToPress`, wonach sich das MSD in einem heißen ausgeführten Cut vor der `false`-Bedingung befindet. Da diese niemals erfüllt werden kann, kann der Cut nur durch die kalte verbotene Nachricht `press` vom PressController pC an Press p verlassen werden. Diese führt zur Terminierung des aktiven MSDs für **R5** und modelliert damit, dass das Sendeverbot für die Nachricht `plateToBelt` nach `press` nicht mehr gelten soll.



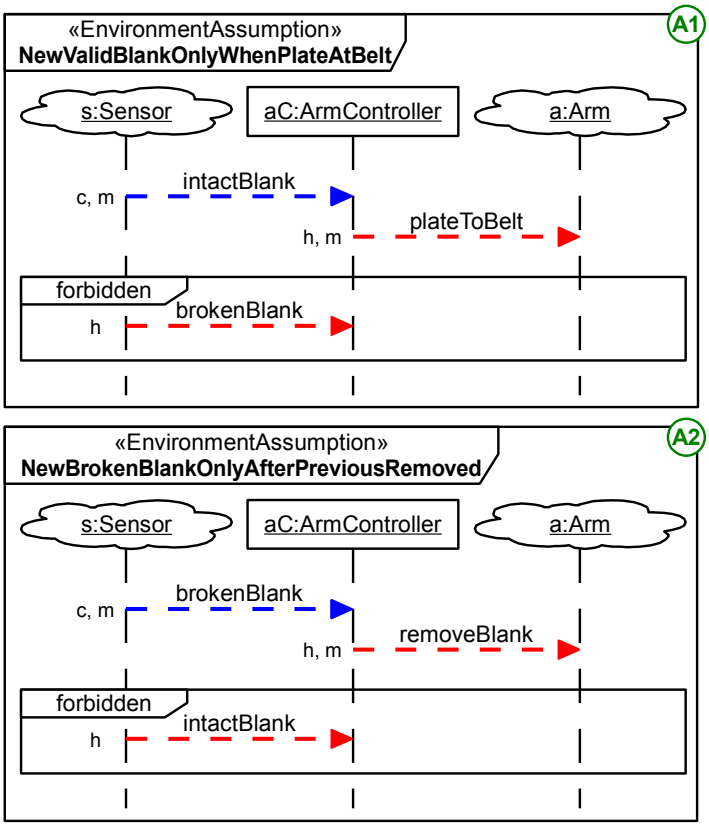
„Wenn das Steuergerät des Arms diesen anweist, einen Rohling zur Presse zu bewegen, dann darf es dem Arm nicht sofort den Befehl erteilen, eine fertige Metallplatte zum Ablageförderband zu bewegen. Dies ist erst dann gestattet, wenn das Steuergerät der Presse diese zum Pressen veranlasst hat.“

Abbildung 3.3: MSD für die Anforderung R5 an die Produktionszelle

3.1.3 Umgebungsannahmen

Um die Anforderungen **R1** und **R4** zu erfüllen, muss das System seine Reaktion auf die Umgebungsnachricht `intactBlank` bzw. `brokenBlank` abschließen, bevor eine dieser beiden Nachrichten erneut auftritt. Die Umgebung kann diese Requirement MSDs also jederzeit verletzen, indem es die Nachrichten in zu kurzen Abständen sendet. Daher sind diese Anforderungen an das System nur unter zusätzlichen Annahmen realisierbar, die nachfolgend beschrieben werden.

Abbildung 3.4 zeigt auf der linken Seite die Assumption MSDs, welche die beiden Annahmen **A1** und **A2** an das Kommunikationsverhalten der Umgebung der Produktionszelle modellieren. Auf der rechten Seite werden dieselben Annahmen jeweils informell in natürlichsprachlichem Text beschrieben. Beide Annahmen haben im Prinzip dieselbe Struktur: Eine Umgebungsnachricht (`intactBlank` oder `brokenBlank`) aktiviert das jeweilige MSD, das sich dann in einem heißen Cut befindet. Die jeweils andere Umgebungsnachricht wird als heiße verbotene Nachricht definiert. Solange das MSD also in diesem Cut bleibt, ist jedes Auftreten einer der beiden Umgebungsnachrichten eine Verletzung der Umgebungsannahmen. Gemäß der Annahmen ist ein Senden beider Umgebungsnachrichten erst wieder nach Beendigung des Assumption MSDs durch die jeweils passende Systemnachricht (`plateToBelt` bzw. `removeBlank`) möglich.



„Wenn der Sensor am Zuführförderband einen neuen intakten Rohling meldet, dann kann angenommen werden, dass kein weiterer (intakter oder defekter) Rohling ankommt, bis das Steuergerät des Roboterarms den Arm angewiesen hat, eine neue fertige Metallplatte auf dem Ablageförderband abzulegen.“

„Wenn der Sensor am Zuführförderband einen neuen defekten Rohling meldet, dann kann angenommen werden, dass kein weiterer (intakter oder defekter) Rohling ankommt, bis das Steuergerät des Roboterarms den Arm angewiesen hat, den defekten Rohling auszusortieren.“

Abbildung 3.4: MSDs für die der Annahmen an die Produktionszelle

Die hier vorgestellten Annahmen sind in dieser nicht zeitbehafteten Variante des Beispiels strenggenommen nicht erforderlich, wenn angenommen wird, dass das System immer beliebig schnell senden kann, ohne dass Umgebungsereignisse auftreten (siehe Abschnitt 2.1.6). In der zeitbehafteten Variante des Beispiels gilt diese implizite Annahme jedoch nur eingeschränkt, wodurch diese explizit modellierten Annahmen dann notwendig werden.

3.1.4 Zusätzlich benötigte Kommunikation

Die durch die Synthese erzeugten Controller für die Systemobjekte ArmController aC und PressController pC dürfen einerseits keine durch die Spezifikation verbotenen Nachrichten senden, werden aber andererseits durch dieselbe Spezifikation zum Senden bestimmter Nachrichten gezwungen.

In bestimmten Fällen kann ein Controller selbst nicht entscheiden, ob er eine Nachricht senden muss oder nicht senden darf – bis er eine zusätzliche, nicht in der MSD-Spezifikation enthaltene Nachricht von einem anderen Controller empfängt. Dieser Abschnitt beschreibt zunächst kurz eine Anforderung, welche ohne zusätzliche Kommunikation erfüllt werden kann und betrachtet dann eine Kombination aus Anforderungen, für die ohne eine zusätzliche Nachricht keine Implementierung möglich ist. Der zweite Fall wurde in Abschnitt 1.2 bereits abstrakter für dasselbe Beispiel beschrieben und wird hier nun konkret auf die MSD-Spezifikation des Beispiels bezogen.

In der gegebenen Spezifikation definiert Anforderung **R4**, dass der ArmController aC nach Erhalt der Nachricht `brokenBlank` irgendwann die Nachricht `removeBlank` sendet. Da `brokenBlank` nicht zur Aktivierung eines anderen MSDs führt und dank Umgebungsannahme **A2** nur einzeln auftritt, darf das Steuergerät aC nach Erhalt von `brokenBlank` in jedem Fall `removeBlank` senden. Ein korrektes Verhalten der Controller ist hier also trivial: Auf jeden Empfang der einen Nachricht wird mit dem Senden der anderen reagiert. Hier wird also keine zusätzliche Kommunikation benötigt.

Ein komplexerer Fall ergibt sich aber durch die Kombination der Anforderungen **R1** und **R5**: Während **R1** vom ArmController aC verlangt, erst `blankToPress` und dann (irgendwann) `plateToBelt` an Arm a zu senden, verbietet **R5** genau diese Sequenz, wenn zwischenzeitlich nicht der PressController pC die Nachricht `press` an die Presse p gesendet hat.

Die Anforderungen können also nur dadurch erfüllt werden, dass aC erst nach der Nachricht `press` selbst die Nachricht `plateToBelt` sendet. Problematisch ist hier jedoch, dass aC die Nachricht `press` weder empfängt noch sendet und daher nicht feststellen kann, ob pC sie bereits gesendet hat. Die Implementierung von ArmController aC kann also nur dann auf diese Nachricht reagieren, wenn sie zuvor durch Empfang einer anderen Nachricht über sie informiert wird.

Würde die MSD-Spezifikation also nach jedem Senden von `press` eine bestimmte Nachricht von pC an aC vorsehen, sodass ein Empfang dieser Nachricht das Senden von `press` impliziert, dann könnte der Controller für aC direkt darauf reagieren. Er könnte dann nach jedem Empfang dieser Nachricht `plateToBelt` senden. Das hier vorgestellte Syntheseverfahren identifiziert solche Fälle und erzeugt in diesen keine (unnötige) zusätzliche Kommunikation.

Die vorliegende MSD-Spezifikation sieht jedoch nicht vor, dass aC nach jedem Senden von `press` über dieses informiert wird. Zu diesem Zweck muss hier daher eine Nachricht an aC gesendet werden, die nicht explizit in der Spezifikation definiert ist. Diese Nachricht kann hier nur vom PressController pC kommen, da dieser im Beispiel das einzige andere Systemobjekt ist. Aus der Spezifikation ist weiterhin ersichtlich, dass pC mit aC kommunizieren kann und dass pC über jedes Senden von `press` informiert ist, da pC selbst der Sender dieser Nachricht ist. Der Syntheseansatz erkennt daher, dass in dieser Situation zusätzliche Kommunikation von pC an aC erforderlich und möglich ist, und erzeugt beide Controller konsistent so, dass eine entsprechende Nachricht zwischen diesen ausgetauscht wird. Der Name dieser Nachricht ist beliebig, sie muss für aC aber von den Nachrichten der Spezifikation unterscheidbar sein.

3.2 Überblick und Einordnung

Dieser Abschnitt bietet einen Überblick über die Lösung des Problems der verteilten Synthese durch das hier vorgestellte Verfahren. Die Eingabe des Verfahrens ist, wie bereits erwähnt wurde, eine MSD-Spezifikation. Der Synthesealgorithmus operiert jedoch nicht direkt auf dieser, sondern er nutzt ein separates Verfahren [BGP13], um schrittweise einen Graphen zu erzeugen, der die globalen Anforderungen der MSD-Spezifikation an das Systemverhalten beschreibt. Anhand dieses Graphen prüft der Synthesealgorithmus periodisch, ob die bisher konstruierten Controller die Anforderungen bereits in allen erreichbaren globalen Zuständen erfüllen oder noch verändert werden müssen.

In Abschnitt 3.2.1 wird zunächst erläutert, wie der Zustandsraum einer MSD-Spezifikation als Graph repräsentiert werden kann. Auf dieser Basis werden in Abschnitt 3.2.2 das Problem der verteilten Synthese und die Kriterien für eine Lösung dieses Problems charakterisiert. Abschnitt 3.2.3 gibt einen abstrakten Überblick über das in dieser Arbeit vorgestellte Syntheseverfahren. Dieses Verfahren wird in Abschnitt 3.2.4 von Vorarbeiten abgegrenzt – insbesondere von einem anderen Syntheseverfahren für MSD-Spezifikationen, das an derselben Fachgruppe wie diese Arbeit entstanden ist.

3.2.1 Die MSD-Spezifikation als Graph

Ziel der Synthese – auch für verteilte Systeme – ist, dass die erzeugte Implementierung ein korrektes globales Systemverhalten bewirkt. Dieses wird durch die MSD-Spezifikation (siehe Abschnitt 2.1) definiert. Bei unterschiedlichen Nachrichtensequenzen im System können im Allgemeinen in einer gegebenen MSD-Spezifikation mehrere verschiedene Konfigurationen von Cuts erreicht werden. Jede dieser Konfigurationen kann als *Zustand* der MSD-Spezifikation aufgefasst werden, von dem aus Nachrichtenereignisse *Transitionen* in andere Zustände auslösen können.

Werden symbolische Lifelines oder Variablen in der MSD-Spezifikation verwendet, so wird ein Zustand zusätzlich zur aktuellen Konfiguration von Cuts durch die Werte aller Variablen und die Bindungen aller Lifelines definiert. Der gesamte durch eine MSD-Spezifikation definierte Zustandsraum kann durch ein Transitionssystem repräsentiert werden, das hier als *Spezifikationsgraph (SG)* bezeichnet wird. Dieses Modell ist die

Grundlage für den in dieser Arbeit beschriebenen Synthesealgorithmus für verteilte Systeme und war bereits die Grundlage für einen zentralen Synthesealgorithmus [GBC+13].

Der SG ist ein Graph, dessen Knoten Zustände (auch: *SG-Zustände*) und dessen Kanten Transitionen (auch: *SG-Transitionen*) sind. Die Zustände repräsentieren jeweils eine Menge aktiver MSDs mit einer bestimmten Konfiguration von Cuts, einer Belegung für alle Variablen und einer Bindung für jede symbolische Lifeline. Der Startzustand ist ein Zustand ohne aktive MSDs. Die Transitionen sind mit Nachrichtenereignissen beschriftet. Wenn eine Transition mit einem Nachrichtenereignis zwischen zwei Zuständen existiert, dann bewirkt das Auftreten dieses Ereignisses im Ausgangszustand (also das Senden der entsprechenden Nachricht) eine Änderung der Cut-Konfiguration der aktiven MSDs zu derjenigen des Folgezustands. Vom Startzustand aus lässt sich daher anhand der Transitionen schrittweise die Konfiguration der Cuts für jeden anderen Zustand ermitteln.

Transitionen, die mit Systemereignissen beschriftet sind, sind *kontrollierbar*, während Transitionen, die mit Umgebungsereignissen beschriftet sind, *unkontrollierbar* sind. Die durch die Synthese erzeugten Controller können nur kontrollierbare Nachrichten senden. Der SG kann als *Spielgraph* [TA99] betrachtet werden, d. h. er repräsentiert ein Spiel des Systems gegen seine Umgebung, welches das System nur durch Erfüllen der Spezifikation gewinnen kann. Dabei kann das System über das Schalten der kontrollierbaren und die Umgebung über das Schalten der unkontrollierbaren Transitionen entscheiden. Auf Grundlage dieser Sichtweise werden in Abschnitt 3.2.2 die Kriterien für eine Lösung des Syntheseproblems definiert.

Abbildung 3.5 zeigt auf der rechten Seite einen Ausschnitt aus dem SG für die MSD-Spezifikation der Produktionszelle aus Abschnitt 3.1. Zustände des SG werden im Text dieser Arbeit zur besseren Unterscheidbarkeit von Zuständen anderer Modelle mit dem Präfix *S.* versehen. Im Startzustand S.0 sind keine MSDs aktiv; das System muss also nichts senden, um die Anforderungen zu erfüllen. Ein solcher Zustand wird als *Zielzustand* bezeichnet. In der Abbildung ist er grün gefärbt und mit einem *G* (für engl. ***g**oal state*) markiert. Zielzustände werden in Abschnitt 3.2.2 näher behandelt.

Das System muss in S.0 zwar nichts senden, aber auf Nachrichten von der Umgebung warten. Daher hat dieser Zustand ausgehende unkontrollierbare Transitionen für die beiden möglichen Umgebungsereignisse `intactBlank` und `brokenBlank` von `s` an `aC`. Für jedes Nachrichtenereignis wird der Name der gesendeten Nachricht angegeben. Darunter sind die beteiligten Kommunikationspartner in der Form `[Sender->Empfänger]` aufgeführt. Unkontrollierbare Transitionen sind gestrichelt eingezeichnet, kontrollierbare durchgezogen.

In Zustand S.2 sind die beiden MSDs für **R1** und **A1** in den links eingezeichneten und mit der Zustandsnummer markierten Cuts aktiv. Als einzige erlaubte Systemnachricht kann `aC` `blankToPress` an `a` senden. Dies führt zur Cut-Konfiguration in Zustand S.3, die ebenfalls in den MSDs eingezeichnet ist. In diesem Zustand existiert nun auch eine Instanz des MSDs zu **R2**. Nun ist `blankAtPress` die einzige erlaubte Systemnachricht, die zu einem noch nicht explorierten Zustand führt. Die Cuts für Zustand S.1 betreffen andere MSDs und sind daher der Übersichtlichkeit halber nicht eingezeichnet.

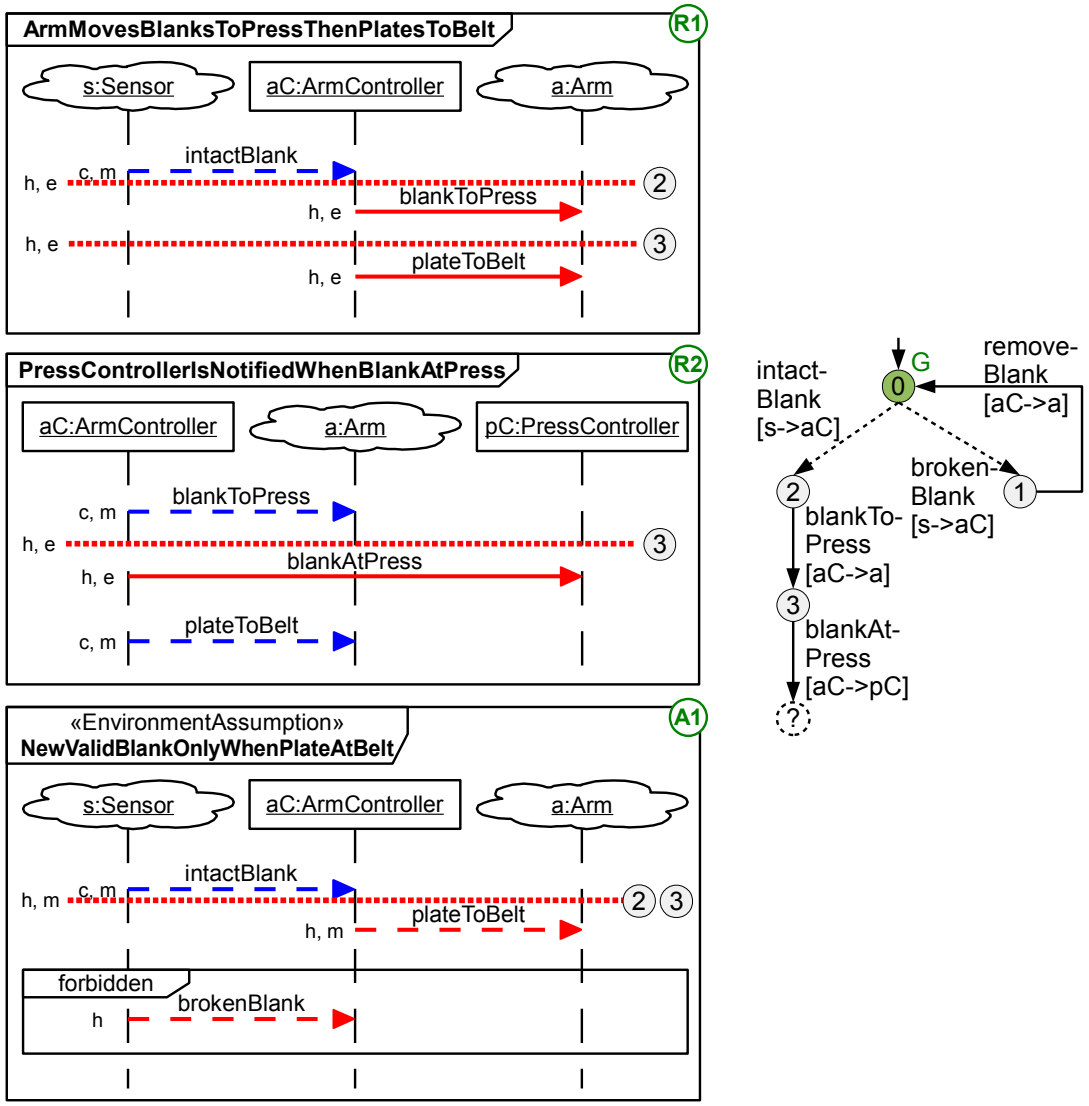


Abbildung 3.5: Ausschnitt aus dem SG für die Produktionszelle (rechts), dazu die Cuts in den Zuständen S.2 und S.3 (links)

3.2.2 Problem und Lösungen der verteilten Synthese

Dieser Abschnitt charakterisiert zunächst das Problem der verteilten Synthese und grenzt es von der globalen Synthese ab. Anschließend wird das Problem der globalen Synthese, wie in der Literatur [BS03] vorgeschlagen, als Zwei-Spieler-Spiel formalisiert. Diese Formalisierung wird in dieser Arbeit auch für die verteilte Synthese als Kriterium für ein korrektes globales Verhalten verwendet. Auf Grundlage dieser Formalisierung wird dann definiert, welche Kriterien eine Lösung der verteilten Synthese erfüllen muss.

Das Problem der verteilten Synthese

Das Problem der *verteilten Synthese* besteht darin, Verhaltensmodelle für eine gegebene Menge an Systemobjekten so zu erzeugen, dass eine gegebene MSD-Spezifikation erfüllt wird. Gemäß Abschnitt 2.2 werden diese Verhaltensmodelle hier als *Controller* und ihre Gesamtheit als *Controllersystem* bezeichnet. Generell müssen für alle Systemobjekte Controller erzeugt werden. Im Fall der Beispiel-Spezifikation aus Abschnitt 3.1 sind dies ArmController und PressController. Wenn ein Controllersystem die Einhaltung der Spezifikation garantiert, dann ist es eine *Lösung* des Problems der verteilten Synthese.

In dieser Arbeit wird das Controllersystem dabei als offenes System betrachtet, d. h. es kann neben den gegebenen Systemobjekten, für die Controller erzeugt werden sollen, im auch Umgebungsobjekte geben, für die keine Controller erzeugt werden sollen (und dürfen). Diese Objekte werden von der Umgebung gesteuert. Ein Controllersystem ist nur dann eine Lösung für das Problem der verteilten Synthese, wenn es die MSD-Spezifikation für beliebiges Umgebungsverhalten einhält. Sofern Umgebungsannahmen definiert sind, ist das Umgebungsverhalten aber nur beliebig im Rahmen dieser Annahmen.

Globale (also nicht-verteilte) Syntheseansätze erzeugen nur ein einziges Verhaltensmodell, einen *globalen Controller*. Die globale Synthese setzt dabei implizit voraus, dass der globale Controller jederzeit sämtliche vorherige Nachrichten kennt, die vom System empfangen oder innerhalb des Systems ausgetauscht werden. Er kann daher nur dann für die Implementierung eines verteilten Systems verwendet werden, wenn er durch sämtliche Objekte über die von diesen gesendeten oder empfangenen Nachrichten unterrichtet wird und er alle Objekte zentral steuern kann. Dies kann aber aufgrund von architekturbedingten Einschränkungen der Kommunikation unmöglich sein. Weiterhin kann der Zeitaufwand der Kommunikation, die zum Informieren des globalen Controllers bzw. zur Steuerung der Objekte erforderlich ist, zur Verletzung von Echtzeitanforderungen führen, wie sie in Kapitel 4 behandelt werden. Selbst wenn eine zentrale Realisierung möglich ist, kann sie aus Gründen schlechterer Performanz impraktikabel sein.

Im Unterschied zum globalen Syntheseproblem muss die verteilte Synthese sicherstellen, dass sich die einzelnen Objekte mittels lokal verfügbarer oder durch Kommunikation ermittelbarer Informationen autonom für eine konkrete Handlungsweise entscheiden können. Jedes Objekt muss diese Entscheidung also auf Basis seines eingeschränkten Wissens über den globalen Gesamtzustand des Systems treffen. Aufgrund globaler Anforderungen können jedoch Abhängigkeiten zwischen den verschiedenen Objekten entstehen, die eine Abstimmung dieser Entscheidungen erfordern. Insbesondere kann ein Objekt auf das Wissen über die Handlungen eines anderen Objekts angewiesen sein, um

selbst korrekt auf diese reagieren zu können. Wie bereits in Abschnitt 3.1.4 motiviert wurde, kann der benötigte Wissensaustausch zusätzliches Kommunikationsverhalten erfordern, das in der Spezifikation nicht explizit vorgeschrieben ist und das die verteilte Synthese daher erzeugen muss.

Lösungen der verteilten Synthese

Wie oben bereits erwähnt wurde, ist eine Lösung der verteilten Synthese ein Controllersystem, das die MSD-Spezifikation erfüllt (vgl. Abschnitte 2.1.6 und 2.2). Es darf also in keinem durch die Spezifikation beschriebenen Zustand Nachrichten senden, die durch diese verboten werden (*Safety*). Weiterhin müssen alle Nachrichten, die die Spezifikation verlangt, irgendwann gesendet werden (*Liveness*). Die Spezifikation ist jedoch in jedem Fall erfüllt, wenn die Umgebung die für sie definierten Annahmen verletzt, selbst bei einer zusätzlichen Verletzung der Anforderungen.

Liveness-Anforderungen werden in MSDs dadurch ausgedrückt, dass Nachrichten als ausgeführt deklariert werden. Jeder ausgeführte Cut muss irgendwann verlassen werden. Wie bereits in Abschnitt 2.1.6 definiert wurde, wird diese Semantik hier durch eine etwas einfachere, aber restriktivere, ersetzt: Um die Liveness-Anforderungen einer MSD-Spezifikation zu erfüllen, muss das System immer wieder einen Zustand erreichen, in dem *kein* aktives Requirement MSD in einem ausgeführten Cut ist. Wir bezeichnen diese speziellen SG-Zustände als *Zielzustände* – wenn in ihnen bisher keine Verletzung der Safety-Anforderungen aufgetreten ist. In diesen Zuständen hat das System keine Verpflichtungen, etwas zu tun, und wartet auf das nächste Umgebungsereignis. Im Beispiel in Abbildung 3.5 ist dies für den Startzustand $\mathsf{S}.0$ der Fall.

Während der Synthese wird eine Strategie für die Auswahl kontrollierbarer Transitionen durch das System erzeugt, sodass immer irgendwann ein Zielzustand erreicht wird (Büchi-Bedingung) [Büc90]. Der Algorithmus muss diese Auswahlstrategie unter der Annahme berechnen, dass die Umgebung (im Rahmen der Umgebungsannahmen) alles tun kann, um das System daran zu hindern, einen Zielzustand zu erreichen. Diese Auswahlstrategie für Transitionen kann auch als *Strategie* des Systems in einem Spiel gegen die Umgebung betrachtet werden [BSL04]. In diesem Zwei-Spieler-Spiel gewinnt das System immer dann, wenn es entweder alle Anforderungen erfüllt oder wenn die Umgebung mindestens eine Annahme verletzt. Andernfalls gewinnt die Umgebung.

Die Strategie muss Safety Violations von Requirement MSDs vermeiden, außer wenn diese zwangsläufig eine Verletzung der in Assumption MSDs spezifizierten Annahmen bedingen. Ist letzteres der Fall, dann kann dieses Verhalten für eine mit den Annahmen konforme Umgebung nicht auftreten und daher im realen System niemals tatsächlich zu einer Verletzung der Anforderungen führen. Daher sind Zustände, in denen die Assumption MSDs bereits durch eine Safety Violation verletzt wurden, ebenfalls Zielzustände, selbst wenn auch die Requirement MSDs verletzt wurden. In diesem speziellen Fall muss das System auch nicht mehr auf Umgebungsereignisse reagieren.

Es ist aber auch dann eine Verletzung der Assumption MSDs, wenn ein Zustand erreicht wird, ab dem – bei Auswahl geeigneter Systemereignisse – immer ein Assumption MSD aktiv ist, dessen Cut ausgeführt ist. Solange die Umgebung nicht veranlasst, dass diese Zustände verlassen werden, muss das System nichts tun. Daher sind auch

diejenigen Zustände Zielzustände, in denen die Anforderungen verletzt wurden, aber mindestens ein Assumption MSD in einem ausgeführten Cut ist. Sofern nur letzteres gilt, die Anforderungen aber nicht verletzt sind, wird der betroffene Zustand zunächst nicht als Zielzustand behandelt. Auf diese Weise muss normalerweise nur ein Systemereignis statt mehrerer Umgebungsereignisse betrachtet werden. Das System hat in diesen Zuständen aber die Wahl, auf die Umgebung zu warten – die irgendwann eine Nachricht senden muss, statt selbst Nachrichten zu senden.

Insgesamt wird eine Gewinnstrategie hier wie folgt definiert [BGS15]:

Definition 3 (Gewinnstrategie, -bedingungen)**.** Eine *Strategie* ist eine Teilmenge von Transitionen eines SG. Eine Strategie ist *gewinnend* (und damit eine *Gewinnstrategie*), wenn die folgenden beiden *Gewinnbedingungen* gelten:

g1 Wenn nur die Transitionen der Strategie verwendet werden, dann enthält der SG keine Deadlocks und keine Zyklen ohne Zielzustände.

g2 Wenn eine Strategie mindestens eine unkontrollierbare ausgehende Transition eines Zustands enthält, dann muss sie auch alle anderen unkontrollierbaren ausgehenden Transitionen dieses Zustands enthalten.

Als Deadlocks gelten hier Zustände, die keine ausgehenden Transitionen in der Strategie enthalten. Zielzustände gelten nur dann als Deadlocks, wenn sie unkontrollierbare ausgehende Transitionen haben, die nicht in der Strategie enthalten sind.

Die Gewinnbedingung **g1** fordert, dass für eine Gewinnstrategie die Büchi-Eigenschaft bezüglich der Zielzustände gelten muss, d. h. es wird garantiert immer (von jeden Zustand aus) ein Zielzustand erreicht, unabhängig vom Verhalten der Umgebung. Die Gewinnbedingung **g2** fordert von einer Gewinnstrategie, dass diese alle möglichen Umgebungsnachrichten (alle Spielzüge der Umgebung) berücksichtigen muss. Insgesamt entspricht eine Gewinnstrategie den Bedingungen aus Abschnitt 2.1.6 für die Implementierung einer MSD-Spezifikation.

Eine *Lösung* für das Problem der verteilten Synthese ist damit ein Controllersystem, welches einer Gewinnstrategie entspricht. Ein Controllersystem entspricht dann einer Gewinnstrategie, wenn es sich durch parallele Komposition der Controller auf einen *globalen Controller* abbilden lässt, der einer Gewinnstrategie entspricht. Ein globaler Controller ist ein Controller, der alle Systemobjekte eines Controllersystems kontrolliert. Dessen Transitionen entsprechen direkt einer Auswahl der kontrollierbaren Transitionen des SG und damit einer Strategie.

Die globale Synthese ist ein Spezialfall der verteilten Synthese, bei der das erzeugte Controllersystem aus einem globalen Controller besteht.

3.2.3 Überblick über das Syntheseverfahren

Der Algorithmus erzeugt für eine gegebene MSD-Spezifikation – sofern möglich – ein Controllersystem, das diese Spezifikation implementiert. Ein Controllersystem implementiert eine MSD-Spezifikation genau dann, wenn es einer Gewinnstrategie (siehe Abschnitt 3.2.2) im SG entspricht, und damit eine Lösung der verteilten Synthese darstellt.

Der Algorithmus erzeugt systematisch *Controllersystem-Kandidaten* (oder kurz: *Kandidaten*), also Controllersysteme, die möglicherweise noch zu einer Lösung erweitert werden können. Diese sind gleichzeitig Kandidaten für eine Gewinnstrategie, da sie eine Teilmenge der Transitionen des SG definieren, die möglicherweise zu einer Gewinnstrategie erweitert werden kann. Initial ist diese Menge von Transitionen leer. Das Hinzufügen einer Transition in einem lokalen Controller impliziert, dass diese Transition in einem oder mehreren SG-Zuständen geschaltet wird. Die entsprechenden Transitionen im SG werden dann Teil der durch den Kandidaten definierten Strategie.

Für die erzeugten Controllersystem-Kandidaten können die erreichbaren SG-Zustände noch die Gewinnbedingungen **g1** und **g2** (siehe Abschnitt 3.2.2) verletzen. Bei Berücksichtigung nur der in der Strategie enthaltenen Transitionen können bestimmte Zustände noch Deadlock-Zustände sein (verletzt **g1**). Andererseits kann die Strategie für bestimmte Zustände noch nicht alle ausgehenden unkontrollierbaren Transitionen enthalten (verletzt **g2**). Solche Zustände, die **g1** oder **g2** verletzten, werden hier als *unimplementiert*, die übrigen als *implementiert* bezeichnet. Um den Kandidaten zu einer gültigen Gewinnstrategie zu vervollständigen, muss der Algorithmus diesen durch Hinzufügen weiterer lokaler Controller-Transitionen erweitern.

Abbildung 3.6 zeigt einen groben Überblick über den Syntheseansatz. Der Ansatz ist im Wesentlichen Gegenbeispiel-getrieben: In jeder Iteration wird für den aktuell betrachteten Kandidaten geprüft, ob dieser eine Lösung ist, d. h. ob er in allen SG-Zuständen die Spezifikation erfüllt. Wenn das der Fall ist, dann terminiert der Algorithmus mit dieser Lösung. Andernfalls werden die unimplementierten Zustände als Gegenbeispiel ermittelt. Auf Grundlage des Gegenbeispiels und basierend auf dem aktuellen Kandidaten wird dann der nächste Kandidat erzeugt, in dem einer der bisher unimplementierten Zustände implementiert wird (also **g1** und **g2** erfüllt). Dazu wird der aktuelle Kandidat durch Hinzufügen von Transitionen und Zuständen erweitert. Die erzeugten Transitionen können auch zusätzliche Kommunikation zum Austausch von Informationen zwischen den Subsystemen definieren, deren lokale Informationen jeweils durch den Algorithmus erfasst werden. Sofern ein SG-Zustand dennoch nicht implementiert werden kann, so wird ein früherer Kandidat als Grundlage der Konstruktion des nächsten Kandidaten genommen. Ist auch das nicht möglich, so terminiert der Algorithmus ohne Lösung. Abschnitt 3.3 konkretisiert dieses Verfahren und erläutert insbesondere, wie lokale Informationen der Subsysteme und deren Austausch modelliert werden.

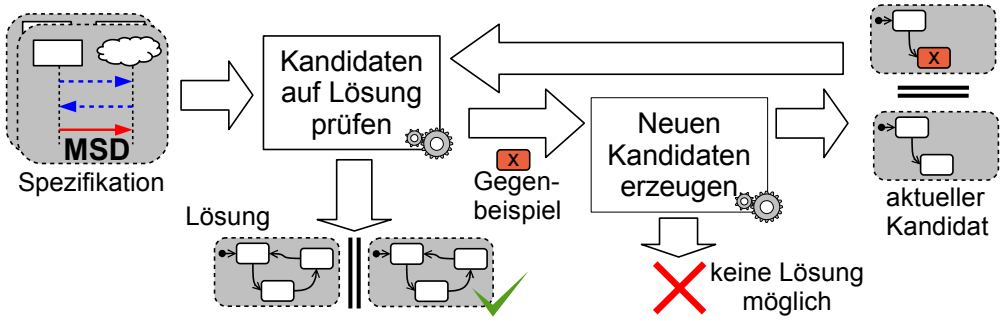


Abbildung 3.6: Abstrakter Überblick über den Syntheseansatz

3.2.4 Abgrenzung zu Vorarbeiten

Dieser Abschnitt grenzt den in dieser Arbeit entwickelten Ansatz von Vorarbeiten ab. Zunächst werden die wesentlichen Unterschiede zu einem anderen Syntheseansatz aus derselben Fachgruppe diskutiert. Anschließend wird begründet, weshalb dieser bestehende Ansatz nicht zur Erzeugung eines Zwischenergebnisses der verteilten Synthese genutzt wird. Schließlich werden weitere Arbeiten eingeordnet, die zum Teil Vorarbeiten darstellen, zum Teil aber auch in Kooperation entstanden sind.

Abgrenzung zur globalen Synthese von Greenyer

In der Dissertation von Greenyer [Gre11], die in derselben Fachgruppe wie die vorliegende Arbeit entstand, wurde bereits ein Syntheseansatz für MSD-Spezifikationen entwickelt. Dieser Ansatz erzeugt jedoch einen globalen Controller und ist daher für verteilte Systeme nicht geeignet: Einerseits benötigt der erzeugte globale Controller vollständige Informationen über den Gesamtzustand des Systems. Diese Informationen liegen jedoch im Allgemeinen nur verteilt vor, da die Objekte jeweils nur über lokale Informationen verfügen. Andererseits muss der globale Controller alle Objekte steuern, selbst wenn diesen eine lokale Entscheidung möglich wäre. Insgesamt wäre ein globaler Controller nur zur Steuerung eines verteilten Systems geeignet, wenn er mit allen Objekten bei vernachlässigbarer Übertragungszeit (in „Nullzeit“) kommunizieren kann.

Anders als bei dieser Arbeit ist das Ziel des Syntheseverfahrens bei Greenyer in erster Linie, widersprüchliche Anforderungen in der MSD-Spezifikation aufzudecken oder deren Abwesenheit nachzuweisen [ABD+14]. Der Controller wird also weniger als Basis für eine spätere Weiterentwicklung, sondern primär als Nachweis für die *Konsistenz* der Spezifikation, also der Widerspruchsfreiheit der Anforderungen, erzeugt. Neben der eigentlichen Synthese wird dort daher auch die interaktive Schritt-für-Schritt-Simulation der MSD-Spezifikation durch ein *Play-out* [HM02a] betrachtet, auch in Kombination mit synthetisierten Controllern [FG12, Gre11]. Eine Weiterentwicklung auf Grundlage des erzeugten Controllers als Implementierungsmodell wird als mögliche zukünftige Arbeit genannt, aber nicht weiter untersucht.

Neben dem Umstand, dass der Syntheseansatz von Greenyer keine verteilten Controller erzeugen kann, ist auch der Syntheseansatz selbst grundsätzlich verschieden zu dem in dieser Arbeit entwickelten. Abbildung 3.7 zeigt einen Überblick über diesen Ansatz. Die MSD-Spezifikation wird auf ein Netzwerk von Timed Game Automata (TGA) [MPS95] abgebildet. Dieses Netzwerk wird so erzeugt, dass es ein Spiel zwischen System und Umgebung modelliert, in dem das System gewinnt, wenn entweder alle Anforderungen erfüllt werden, oder mindestens eine Annahme verletzt wird. Die eigentliche Synthese wird dann durch das Werkzeug Uppaal Tiga[1] [CDF+05] durchgeführt: Dieses sucht eine Gewinnstrategie für das System (siehe Abschnitt 3.2.2), also eine Beschreibung, wie das System alle Anforderungen für jede Umgebung erfüllen kann, die die Annahmen erfüllt. Diese Gewinnstrategie entspricht einem globalen Controller.

Es ist denkbar, dass sich auf Grundlage von TGA und unter Verwendung von Uppaal Tiga auch ein verteilter Syntheseansatz umsetzen lässt. Uppaal Tiga berechnet jedoch

[1] `http://people.cs.aau.dk/~adavid/tiga/`

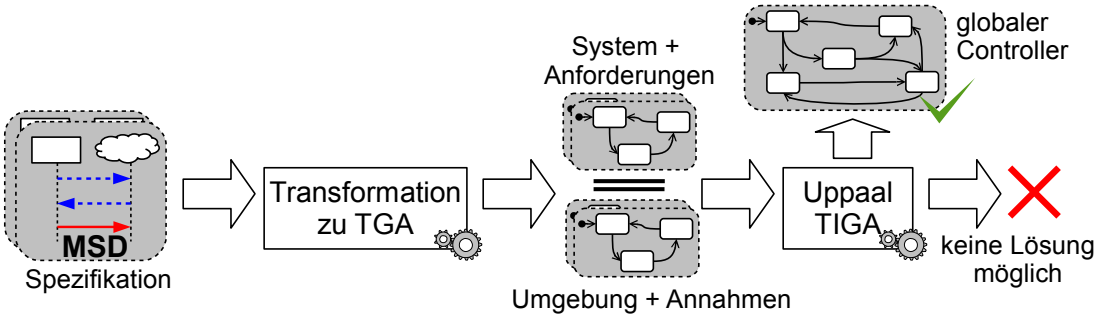


Abbildung 3.7: Übersicht über die TGA-basierte globale Synthese

nur eine globale Gewinnstrategie. Während die Verwendung des Werkzeugs für eine globale Synthese nur eine relativ direkte Abbildung der MSDs auf TGA erfordert, müssten die TGA für eine verteilte Synthese zusätzlich alle sich durch die Verteilung ergebenden Einschränkungen kodieren. Dies wurde in dieser Arbeit jedoch nicht weiter untersucht, da die Verwendung von TGA bzw. Uppaal Tiga einige grundsätzliche Nachteile hat. Der Hauptnachteil ist, dass TGA in Uppaal Tiga nicht dynamisch instanziiert werden können, wodurch MSDs generell nur eingeschränkt abgebildet werden können, da von diesen zur Laufzeit beliebig viele Instanzen erzeugt werden können.

In der Abbildung von MSDs auf TGA nach Greenyer wird für jedes MSD nur ein TGA erzeugt, der diesem entspricht. Daher kann es für jedes MSD nur eine Instanz, also ein aktives MSD geben. Tritt die minimale Nachricht desselben MSDs erneut auf, so wird keine zweite Instanz erzeugt, sondern die Nachricht wird ignoriert – sofern sie keine Violation verursacht. Diese alternative Semantik für MSDs wird als „iterative Interpretation“ bezeichnet [HM03], während die Variante mit beliebig vielen Instanzen je MSD als „invariante Interpretation“ bezeichnet wird. Die in dieser Arbeit verwendete MSD-Semantik entspricht der invarianten Interpretation (vgl. Abschnitt 2.1.2) und kann daher durch die Abbildung auf TGA nicht korrekt umgesetzt werden.

Weiterhin können symbolische Lifelines (siehe Abschnitt 2.1.4) nur über einen Umweg unterstützt werden: Da für jede mögliche Bindung von Lifelines an Objekte ein eigenes aktives MSD erzeugt werden kann, TGA in Uppaal Tiga aber nicht dynamisch instanziiert werden können, muss für jede mögliche Bindung vorab ein TGA erzeugt werden. Werden wesentlich mehr Instanzen angenommen, als tatsächlich im System vorhanden sind, so beeinträchtigen die zahlreichen überflüssigen Automaten die Performanz des Verfahrens. Dieses Vorgehen funktioniert grundsätzlich nicht, wenn die maximal mögliche Anzahl von Instanzen nicht bekannt ist.

Ein weiteres Problem des bestehenden Syntheseansatzes ist, dass dieser nicht dieselbe Semantik umsetzt wie die Schritt-für-Schritt-Simulation von MSDs durch ein Play-out (s. o.), die Bestandteil derselben Werkzeugumgebung ist [Gre11]. Dadurch können Entwickler nach Durchführen der Simulation nicht sicher sein, dass ein für dieselbe MSD-Spezifikation erzeugter Controller das in der Simulation beobachtete Verhalten umsetzt. Da Simulation und Synthese zudem unterschiedliche Modellelemente unterstützen, kann je nach den in der Spezifikation verwendeten Elementen im Einzelfall nur entweder die

Synthese oder die Simulation anwendbar sein. Um dies zu vermeiden, wurde der in dieser Arbeit entwickelte Syntheseansatz auf Grundlage derselben Interpretation von MSDs entwickelt wie die Simulation.

Argumente gegen eine nachträgliche Verteilung globaler Controller

Ein möglicher Ansatz zur Erzeugung verteilter Controller wäre es, erst einen globalen Controller zu erzeugen und diesen dann zu verteilen. Dies hätte den Vorteil der Wiederverwendung des bestehenden Syntheseverfahrens zur globalen Synthese. Ansätze zur Verteilung globaler Controller existieren bereits [HK02, HB10] und werden in Kapitel 7 kurz diskutiert. Diese Herangehensweise hat jedoch ein grundsätzliches Problem: Die globale Synthese kann einen globalen Controller erzeugen, dessen Verteilung zusätzliche Nachrichten erfordert, die im realen System nicht gesendet werden können.

Das Problem eines nicht verteilbaren globalen Controllers kann aus unterschiedlichen Gründen auftreten. Im Falle von Echtzeitsystemen kann es sein, dass die Übermittlung der zusätzlichen Nachrichten so lange dauert, dass zwangsläufig Echtzeitanforderungen verletzt werden. Auch bei nicht echtzeitkritischen Systemen können architekturbedingte Einschränkungen dazu führen, dass einige Subsysteme nicht miteinander kommunizieren können – beispielsweise fehlende physikalische Netzwerkverbindungen oder eine zu geringe Reichweite eines drahtlosen Netzwerks.

Selbst wenn eine verteilte Implementierung der gegebenen Spezifikation *grundsätzlich* möglich ist, kann die Synthese einen nicht verteilbaren globalen Controller erzeugen. Dann gibt es zwar *grundsätzlich* globale Controller für dieselbe Spezifikation, die unter Einhaltung der Echtzeitanforderungen und der architekturbedingten Einschränkungen verteilt werden können. Ob ein solcher bei einer gegebenen Ausführung der globalen Synthese erzeugt wird ist jedoch Glückssache, da diese dessen Verteilbarkeit nicht berücksichtigt. Zwar könnte man bei Fehlschlag der Verteilung eines globalen Controllers diesen verwerfen und die Synthese erneut durchführen – es gibt aber keine Garantie, dass dann ein anderer Controller erzeugt wird.

Aus diesem Grund wurde in dieser Arbeit kein Ansatz zur Verteilung globaler Controller entwickelt, sondern ein neuer Syntheseansatz, der direkt verteilte Controller erzeugt. Dabei erfolgt die Erzeugung des lokalen Verhaltens der Controller gleichzeitig zur Erzeugung des globalen Verhaltens. Wird dann festgestellt, dass das aktuelle globale Verhalten nicht tatsächlich durch die lokalen Controller umgesetzt werden kann, so kann dieses globale Verhalten direkt verworfen oder modifiziert werden – ohne erst eine komplette globale Synthese durchführen zu müssen.

Einordnung der in Kooperation entstandenen Ergebnisse

Um eine gemeinsame Basis für die Schritt-für-Schritt-Simulation von MSDs und die Synthese zu schaffen, wurde im Rahmen des ScenarioTools-Projekts[2] unter Mitwirkung des Autors dieser Arbeit die „MSD-Runtime“, eine neue Laufzeitumgebung für MSDs, entwickelt [BGP13]. Dieser wird sowohl für die Synthese als auch für die Simulation genutzt, um schrittweise das gemäß der Spezifikation erlaubte globale Systemverhalten zu

[2] `http://www.scenariotools.org`

ermitteln. Diese gemeinsame Basis stellt einerseits sicher, dass die umgesetzte Semantik der MSDs in beiden Fällen dieselbe ist, und reduziert andererseits insgesamt den Implementierungs- und Wartungsaufwand, da der betreffende Anteil der Software nicht doppelt implementiert werden muss. Der in dieser Arbeit vorgestellte Syntheseansatz nutzt die MSD-Runtime zur Erzeugung des SG (siehe Abschnitt 3.2.1).

Als erster Schritt in Richtung eines verteilten Syntheseansatzes auf Basis der MSD-Runtime wurde im ScenarioTools-Projekt zunächst ein neuer Syntheseansatz zur Erzeugung globaler Controller entwickelt [GBC+13]. Abbildung 3.8 zeigt einen groben Überblick über dieses Verfahren. Wie im hier beschriebenen Ansatz wird dort der Zustandsraum der Spezifikation in Form des SG durchsucht. Dieser wird unter Verwendung der MSD-Runtime schrittweise aus der MSD-Spezifikation erzeugt.

Der globale Controller wird jedoch bei diesem globalen Syntheseansatz, anders als die verteilten Controller in der verteilten Synthese, nicht separat zum Zustandsraum der Spezifikation erzeugt, sondern besteht immer aus einer Teilmenge der SG-Zustände und -Transitionen. Dadurch ist die Struktur des Algorithmus deutlich einfacher, weil die Verwaltung der einzelnen Controller entfällt. Da die globale Synthese vollständige Informiertheit des globalen Controllers über den Zustand aller Subsysteme voraussetzt, muss auch kein lokales Wissen der Subsysteme modelliert werden. Ein Einfügen zusätzlicher, nicht in der MSD-Spezifikation definierter Nachrichten ist aus demselben Grund ebenfalls nicht erforderlich.

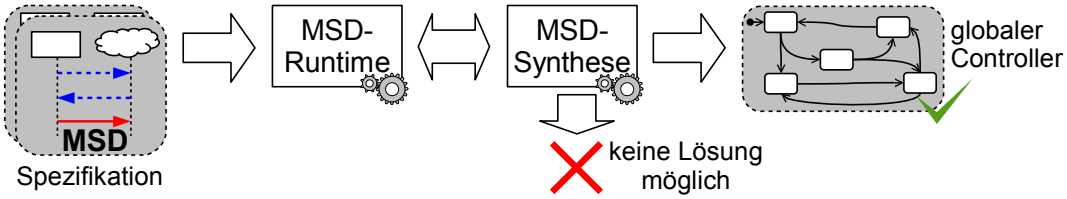


Abbildung 3.8: Übersicht über die kooperativ entwickelte globale Synthese

Insgesamt ist die Struktur des neuen Algorithmus zur globalen Synthese stark auf diesen einfacheren Fall ausgelegt und erlaubt keine einfache Erweiterung zu einem verteilten Syntheseansatz. Der in dieser Arbeit vorgestellte Algorithmus ist insoweit zu dem in Kooperation entwickelten globalen Ansatz ähnlich, als ebenfalls die MSD-Runtime zur Erzeugung eines SG genutzt wird, um in diesem eine Gewinnstrategie zu suchen. Im verteilten Fall kann diese Gewinnstrategie jedoch nur indirekt durch Erzeugung der lokalen Controller und unter Beachtung derer lokalen Informationen gebildet werden.

3.3 Ein Algorithmus für verteilte Synthese

Dieser Abschnitt stellt den Synthesealgorithmus für verteilte Systeme vor. Dabei werden die im vorhergehenden Abschnitt eingeführten Begriffe und Konzepte vorausgesetzt. Abbildung 3.9 zeigt einen Überblick über das Syntheseverfahren. Die Abbildung zeigt die drei wichtigsten Modelle, auf denen der Algorithmus arbeitet: Den Kandidatengraphen, den jeweils aktuellen Controllersystem-Kandidaten und den Spezifikationsgraphen.

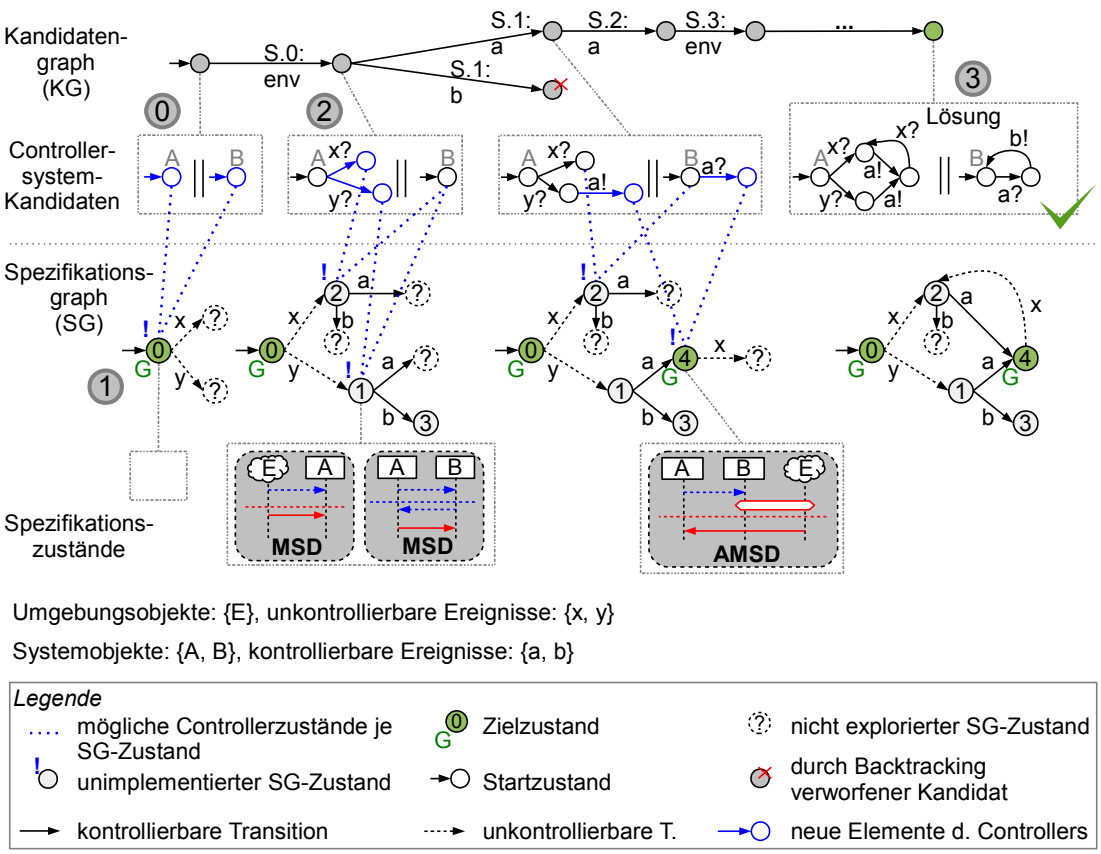


Abbildung 3.9: Überblick über den Ablauf des Syntheseverfahrens

Controllersystem-Kandidaten (oder einfach *Kandidaten*) sind Controllersysteme, die möglicherweise zu einer korrekten Implementierung des verteilten Systems erweitert werden können (siehe Abschnitt 3.2.3). Jeder Controllersystem-Kandidat enthält für jedes Systemobjekt (hier A und B) einen vorläufigen lokalen Controller.

Der *Kandidatengraph (KG)* speichert alle bereits erzeugten Kandidaten (Knoten) und durch welche Erweiterungen gegenüber früheren Kandidaten diese entstanden sind (Kanten). Durch den KG kann somit die erneute Erzeugung von bereits verworfenen Kandidaten vermieden werden.

Der *Spezifikationsgraph (SG)* modelliert den Zustandsraum der Spezifikation (siehe Abschnitt 3.2.1). Dieser ist möglicherweise erst teilweise exploriert worden und kann noch Transitionen enthalten, die zu noch unentdeckten Zuständen führen, welche in Abbildung 3.9 mit „?“ markiert sind.

Abschnitt 3.3.1 erläutert anhand von Abbildung 3.9 den Ablauf des Syntheseverfahrens. In Abschnitt 3.3.2 werden als Kernaspekte des Verfahrens die Modellierung des Wissens der Objekte und das Hinzufügen von Synchronisationsnachrichten zum Austausch dieses Wissens genauer behandelt. Die Abschnitte 3.3.3 und 3.3.4 beschreiben die wesentlichen Schritte des Verfahrens im Einzelnen – die Suche von unimplementier-

ten Zuständen als Gegenbeispiele für eine Lösung und die Erzeugung neuer Kandidaten. Die Korrektheit des Algorithmus wird in Abschnitt 3.3.5 informell begründet. Anhand der in Abschnitt 3.1 beschriebenen MSD-Spezifikation der Produktionszelle wird eine Beispielausführung des Algorithmus in Abschnitt 3.3.6 diskutiert. Abschließend wird in Abschnitt 3.3.7 ein komplexerer Spezialfall der Ausführung des Algorithmus anhand einer geringfügig erweiterten Variante des Beispiels betrachtet.

3.3.1 Ablauf der Synthese

In Abbildung 3.9 werden die Zwischenergebnisse der wesentlichen Schritte des Algorithmus dargestellt. Anhand dieser wird hier der Ablauf erläutert, der in Abschnitt 3.2.3 bereits abstrakter beschrieben wurde und in Abbildung 3.6 schematisch dargestellt ist.

Im ersten Schritt erzeugt der Algorithmus den initialen Kandidaten **(0)**. Alle Controller dieses initialen Kandidaten enthalten jeweils nur einen Startzustand. Zu Beginn der Synthese enthält der KG nur diesen Kandidaten. Für Umgebungsobjekte werden keine Controller erzeugt. Für diese wird hier angenommen, dass sie jede beliebige Sequenz von Umgebungsereignissen senden können – sofern sie nicht den explizit modellierten Annahmen widerspricht.

In jeder Iteration betrachtet der Algorithmus einen *aktuellen Kandidaten*. Der erste von diesen ist der initiale Kandidat. Für den aktuellen Kandidaten werden alle Zustände des SG ermittelt, die erreichbar sind, wenn die Systemobjekte nur Nachrichten entsprechend der jeweiligen Controller des Kandidaten senden. Der Algorithmus prüft dann, ob einer dieser erreichbaren SG-Zustände die Gewinnbedingungen (siehe Abschnitt 3.2.2) verletzt **(1)**. Dieser Schritt wird in Abschnitt 3.3.3 genauer erläutert.

Wir nennen SG-Zustände, welche die Gewinnbedingungen verletzen, *unimplementierte Zustände*. Existiert kein solcher Zustand, dann terminiert der Algorithmus und gibt den aktuellen Kandidaten zurück **(3)**. Da für diesen Kandidaten alle erreichbaren SG-Zustände implementiert sind, erfüllt er die Gewinnbedingungen. Er entspricht damit einer Gewinnstrategie, also einer Lösung des Problems der verteilten Synthese.

Gibt es jedoch unimplementierte SG-Zustände für den aktuellen Kandidaten, so erweitert der Algorithmus den KG durch Konstruktion eines neuen Kandidaten **(2)**. Dieser Schritt wird in Abschnitt 3.3.4 genauer erläutert. Der neue Kandidat wird auf Basis einer Kopie des aktuellen Kandidaten erzeugt. Den Controllern dieser Kopie werden Transitionen so hinzugefügt, dass mindestens einer der bisher unimplementierten SG-Zustände nun die Gewinnbedingungen erfüllt und damit *implementiert* ist. Dazu kann es erforderlich sein, zusätzliche Kommunikation zum Informationsaustausch der Subsysteme zu erzeugen, wie in Abschnitt 3.3.2 erläutert und in Abschnitt 3.3.4 konkretisiert wird. Bei allen Erweiterungen vermeidet es der Algorithmus, Zyklen im SG ohne Zielzustand zu schließen, da dies die Gewinnbedingung **g1** verletzen würde.

Sofern eine solche Erweiterung möglich ist, wird die nächste Iteration des Algorithmus mit dem neuen Kandidaten als aktuellen Kandidaten durchgeführt. Kann jedoch mindestens ein SG-Zustand nicht durch Erweiterung des Kandidaten implementiert werden, so kann dieser Kandidat keine Basis für eine Lösung sein. In diesem Fall führt der Algorithmus ein Backtracking durch, d. h. er verwirft den aktuellen Kandidaten und kehrt zu einem anderen bereits erzeugten (aber noch nicht verworfenen) Kandidaten zurück.

Er versucht dann, basierend auf diesem, einen neuen Kandidaten zu erzeugen. Wenn der Algorithmus jedoch alle möglichen Erweiterungen der existierenden Kandidaten erfolglos versucht hat, dann terminiert er ohne eine Lösung. Insgesamt ist der Algorithmus eine spezielle Form einer Tiefensuche im Kandidatengraph.

3.3.2 Modellierung der lokalen Informationen der Controller und Informationsaustausch durch Synchronisationsnachrichten

Zur Erfüllung einer MSD-Spezifikation müssen sich die lokalen Controller oft in unterschiedlichen SG-Zuständen unterschiedlich verhalten. In diesen Fällen ist es wichtig, ob der betreffende Controller diese Zustände überhaupt voneinander unterscheiden kann. Um dies zu modellieren, legt der Algorithmus *Korrespondenzbeziehungen* zwischen den explorierten SG-Zuständen und den lokalen Controllerzuständen an. Diese Korrespondenzbeziehungen (in Abbildung 3.9 durch blau gepunktete Linien repräsentiert) modellieren, in welchen möglichen (globalen) SG-Zuständen das System aus Sicht eines Controllers sein kann, während er in einem bestimmten (lokalen) Controllerzustand ist. Dadurch modellieren sie gleichzeitig die möglichen Zustände aller Controller in einem bestimmten SG-Zustand.

Da in typischen Systemen nicht jeder lokale Controller jede mögliche Nachricht senden oder empfangen kann, gibt es im Allgemeinen Nachrichten (bzw. entsprechende Ereignisse), die ein Controller nicht beobachten kann. Für diese Nachrichten werden bei der Synthese auch keine Transitionen im betreffenden Controller angelegt, da sie ohnehin nie geschaltet würden. Der Controller bleibt beim Auftreten des entsprechenden Nachrichtenereignisses also in demselben Zustand. Daher kann ein Controllerzustand im Allgemeinen zu mehreren SG-Zuständen korrespondieren. Wie in Abschnitt 3.3.3 erläutert wird, können umgekehrt auch mehrere lokale Controllerzustände zu demselben SG-Zustand korrespondieren.

Ein Controller hat immer dann unzureichende Informationen, wenn er sich in mehreren SG-Zuständen unterschiedlich verhalten – also unterschiedliche Nachrichten senden – muss, diese aber nicht unterscheiden kann. Der Algorithmus stellt dies dadurch fest, dass ein Controllerzustand dann zu mehreren SG-Zuständen korrespondiert und der Controller eine Nachricht nur in einigen von diesen senden darf. In solchen Controllerzuständen würde das Hinzufügen einer ausgehenden Transition, die diese Nachricht sendet, ein Senden auch in SG-Zuständen implizieren, welche das nicht erlauben.

Ein anderer Controller kann aber durchaus in der Lage sein, die SG-Zustände, die ein Senden verbieten, von denen zu unterscheiden, die es erlauben. Um diese Informationen zu dem Controller zu übertragen, der sie benötigt, fügt der Algorithmus bei der Erzeugung eines neuen Kandidaten den Controllern Transitionen hinzu, die zusätzliche, nicht in der Spezifikation vorgesehene Nachrichten zwischen den Controllern austauschen. Diese zusätzlichen Nachrichten werden im Folgenden als *Synchronisationsnachrichten* bezeichnet. Nach Übermittlung der Synchronisationsnachrichten hat der empfangende Controller dieselben Informationen wie der oder die Sender und kann konform zur Spezifikation agieren. Der Algorithmus fügt die Transitionen für diese Synchronisationsnachrichten jedoch nur dann hinzu, wenn es keine andere Möglichkeit gibt, einen unimplementierten SG-Zustand zu implementieren.

3.3.3 Berechnung der erreichbaren unimplementierten SG-Zustände

Die Identifikation der für einen Kandidaten erreichbaren unimplementierten SG-Zustände erfolgt in zwei Schritten: Zunächst wird die Erreichbarkeit der SG-Zustände für den Fall ermittelt, dass das System nur im aktuellen Kandidaten definierte Nachrichten sendet. Als zweiter Schritt wird für die erreichbaren SG-Zustände durch Abgleich mit den Gewinnbedingungen nach Abschnitt 3.2.2 bestimmt, ob diese implementiert sind.

Sowohl die Berechnung erreichbarer SG-Zustände, als auch die Prüfung auf Implementiertheit muss nicht in jeder Iteration vollständig neu durchgeführt werden. Es genügt, diejenigen SG-Zustände zu betrachten, die durch die letzte Erweiterung des Controllersystems neu erreicht werden können bzw. deren Implementiertheit sich durch diese Erweiterung ändert.

Identifikation der erreichbaren SG-Zustände und Berechnung der Zustandskorrespondenzen

Wie oben beschrieben wurde, werden die lokalen Informationen der Controller über die möglichen aktuellen SG-Zustände durch Zustandskorrespondenzen modelliert. Diese werden im Zuge der Berechnung der erreichbaren SG-Zustände ebenfalls ermittelt.

Wenn ein Controller eine SG-Transition nicht beobachten kann, dann kann er den Ausgangs- und den Folgezustand der Transition nicht voneinander unterscheiden. Daher können, während der Controller in demselben Zustand verbleibt, mehrere SG-Zustände durchlaufen werden. Umgekehrt können aber durch die zusätzlich eingefügten Transitionen für Synchronisationsnachrichten mehrere Controllerzustände für denselben SG-Zustand durchlaufen werden. Von den möglichen Kombinationen von Controllerzuständen und SG-Zuständen hängt ab, ob die SG-Zustände implementiert sind, was für den Algorithmus von zentraler Bedeutung ist (vgl. Abschnitt 3.3.1).

Der Synthesealgorithmus benötigt daher ein explizites Modell dieser möglichen Kombinationen. Wir nennen einen SG-Zustand s und einen Controllerzustand c *korrespondierend*, wenn es eine Folge von Ereignissen gibt, nach der die Spezifikation in s ist, während der Controller von c, nach einer der Ereignisfolge entsprechenden Ausführung in Kombination mit den anderen Controllern und der Umgebung, in c ist. Eine der Ereignisfolge entsprechende Ausführung in diesem Sinne ist eine Ausführung, bei welcher der Controller jeweils die Nachrichten sendet bzw. empfängt, die durch die Ereignisfolge vorgegeben sind. Wir nennen eine Controllertransition t_c und eine SG-Transition t_s *korrespondierend*, wenn die Ausgangszustände korrespondierend sind und beide Transitionen mit demselben Ereignis beschriftet sind. Der Algorithmus berechnet diese Korrespondenzen als Relation `corr` korrespondierender Zustände eines SG S (mit Startzustand $S.0$) für einen Controller C (mit Startzustand $C.0$) nach der folgenden Definition [BGS15]:

1. $(C.0, S.0) \in corr$.
2. Für Transition $s \xrightarrow{e} s'$ in S, Ereignis e unbeobachtbar für C:
   $(c, s) \in corr \Rightarrow (c, s') \in corr$.
3. Für Transition $s \xrightarrow{e} s'$ in S, Ereignis e beobachtbar für C, $\exists\ c \xrightarrow{e} c'$ in C:
   $(c, s) \in corr \Rightarrow (c', s') \in corr$.

Wenn ein Controllerzustand zu mehreren SG-Zuständen korrespondiert und der Controller in diesem Zustand eine Nachricht sendet, die Ereignis e entspricht, dann sendet der Controller e in allen diesen SG-Zuständen. Dies betrifft auch SG-Zustände, die vom Algorithmus erst erreicht werden, nachdem die Controllertransition für Ereignis e bereits erzeugt wurde. Bevor diese SG-Zustände erreicht werden, korrespondieren sie zwar noch nicht zum Ausgangszustand der Controllertransition, sie können aber bei ihrem Erreichen durch die induktive Definition von `corr` nachträglich korrespondierend zu diesem Controllerzustand werden.

Wie nachfolgend in Abschnitt 3.3.4 beschrieben wird, kann der Algorithmus den Controllern Transitionen hinzufügen, die Nachrichten senden oder empfangen, mittels derer die Controller synchronisiert werden. Diese Nachrichten sind in der MSD-Spezifikation noch nicht definiert und haben daher auch keine Entsprechung zu SG-Transitionen. Der Sender einer solchen *Synchronisationsnachricht* hat dann eine sendende Transition, die mit einem *Synchronisationsereignis* beschriftet ist, der Empfänger eine mit demselben Ereignis beschriftete empfangende Transition.

Der Sender einer Synchronisationsnachricht überträgt seine Informationen über die möglichen globalen SG-Zustände an den Empfänger. Umgekehrt überträgt der Empfänger seine Informationen aber nicht an den Sender: Da wir annehmen, dass ein Empfänger nicht das Senden von Nachrichten an ihn blockieren kann, kann der Sender aus dem erfolgreichen Senden nichts über den lokalen Zustand des Empfängers oder den möglichen globalen Zustand schließen.

Der Folgezustand der Transition des Senders einer Synchronisationsnachricht hat daher dieselben korrespondierenden SG-Zustände wie der Ausgangszustand. Für den Empfänger der Synchronisationsnachricht korrespondiert der Folgezustand der empfangenden Transition jedoch nur zu denjenigen SG-Zuständen, die zu den Ausgangszuständen *beider* an der Synchronisation beteiligter Transitionen korrespondieren.

Um Transitionen mit Synchronisationsnachrichten zu berücksichtigen, wird die Definition der Relation `corr` wie folgt erweitert [BGS15]:

4. Für eine sendende Transition mit Synchronisationsnachricht $c \xrightarrow{synch!} c'$ in C: $(c, s) \in corr \Rightarrow (c', s) \in corr$.

5. Für eine Transition $c \xrightarrow{synch?} c'$ in C, die eine Synchronisationsnachricht empfängt, die von Transition $c2 \xrightarrow{synch!} c2'$ in einem anderen Controller $C2$ gesendet wurde: $(c, s) \in corr \wedge (c2, s) \in corr \Rightarrow (c', s) \in corr$

Für einen gegebenen Kandidaten nennen wir einen SG-Zustand *erreichbar*, wenn dieser in der parallelen Komposition dieses Kandidaten mit einer zu den Annahmen konformen Umgebung erreicht werden kann. Dies sind alle SG-Zustände, die zu mindestens einem Controllerzustand des Kandidaten korrespondieren.

Prüfung erreichbarer SG-Zustände auf Implementiertheit

Die Implementiertheit von SG-Zuständen hängt von der Implementiertheit ihrer ausgehenden Transitionen ab. Wir nennen eine SG-Transition *implementiert*, wenn ihr

Ereignis implementiert ist. Daher wird nachfolgend zuerst die Implementiertheit von Ereignissen definiert und anschließend die Implementiertheit von SG-Zuständen.

Definition 4 (Implementierte kontrollierbare Ereignisse)**.** Ein *kontrollierbares* Ereignis e ist *implementiert* in einem SG-Zustand s, wenn der Controller des sendenden Objekts e in s sendet. Dazu muss er in allen zu s korrespondierenden Zuständen entsprechende ausgehende Transitionen enthalten – oder Transitionen, die Synchronisationsnachrichten empfangen, und die schließlich zu einem sendenden Zustand führen. Wird e von einem Systemobjekt empfangen, so gilt zusätzlich, dass der Controller des Empfängers eine empfangende Transition für e in *allen* zu s korrespondierenden Zuständen definieren muss.

Sind in einem SG-Zustand mehrere kontrollierbare Ereignisse implementiert, so kann das System jedes von diesen senden, hat aber keine Kontrolle darüber, welches tatsächlich zuerst gesendet wird. Dies liegt daran, dass in einem solchen Fall mehrere Controller unabhängig voneinander senden können. Es müssen somit alle möglichen Reihenfolgen der nebenläufig gesendeten Nachrichten und die entsprechenden Transitionen im SG betrachtet werden.

Definition 5 (Implementierte unkontrollierbare Ereignisse)**.** Ein *unkontrollierbares* Ereignis e ist *implementiert* in einem SG-Zustand s, wenn das empfangende Objekt für e entweder ein Umgebungsobjekt ist, oder sein Controller eine empfangende Transition für e in *allen* zu s korrespondierenden Zuständen definiert.

In Zielzuständen ist das System entweder nicht in einem ausgeführten Cut, die Umgebungsannahmen wurden bereits verletzt oder die Umgebung muss noch Nachrichten senden (siehe Abschnitt 3.2.2). Das System riskiert in diesen Zuständen also durch Abwarten keine Verletzung der Anforderungen durch Verweilen in einem ausgeführten Cut. Da das System in Zielzuständen daher keine Nachrichten senden muss, brauchen in diesen SG-Zuständen keine ausgehenden kontrollierbaren Transitionen implementiert sein. Das System muss jedoch auf alle möglichen Nachrichten der Umgebung reagieren und daher die ausgehenden unkontrollierbaren Transitionen implementieren – außer wenn die Annahmen bereits verletzt wurden.

Definition 6 (Implementierte Zielzustände)**.** Ein *Zielzustand* ist dann *implementiert*, wenn alle seine ausgehenden unkontrollierbaren Transitionen implementiert sind (siehe Gewinnbedingung **g2** in Abschnitt 3.2.2). Ein Zielzustand ist in jedem Fall auch dann implementiert, wenn in ihm die Umgebungsannahmen bereits verletzt sind. Schließlich ist ein Zielzustand auch dann implementiert, wenn er nach den Kriterien für nicht-Zielzustände implementiert ist. Er zählt dann aber im Sinne anderer Definitionen nicht mehr als Zielzustand.

In SG-Zuständen, die keine Zielzustände sind, muss das System sicherstellen, dass die Anforderungen nicht durch Verweilen in einem ausgeführten Cut verletzt werden. Diese Zustände können durch das Senden mindestens einer Systemnachricht implementiert werden. Es können aber auch Fälle auftreten, in denen das System zwar keine Nachrichten senden kann, aber es mindestens ein Assumption MSD in einem ausgeführten Cut

gibt. In diesen Fällen ist die Umgebung gezwungen, zur Erfüllung der Annahmen eine oder mehrere Nachrichten zu senden. Daher kann das System in diesen Fällen auf die Umgebung warten ohne eine Verletzung von Liveness-Anforderungen zu riskieren.

Definition 7 (Implementierte normale SG-Zustände)**.** Ein SG-Zustand, der *kein Zielzustand* ist, ist *implementiert*, wenn dieser mindestens eine ausgehende kontrollierbare Transition hat, die implementiert ist. Ein solcher SG-Zustand ist aber auch dann implementiert, wenn es ein Assumption MSD in einem ausgeführten Cut gibt und alle ausgehenden unkontrollierbaren Transitionen implementiert sind.

Ein Kandidat ist nur dann eine Lösung, wenn für diesen alle erreichbaren SG-Zustände implementiert sind. Durch die hier gegebenen Definitionen für Implementiertheit ist sichergestellt, dass die Strategie, die einer Lösung entspricht, keine Deadlock-Zustände enthält (siehe Bedingung **g1** in Abschnitt 3.2.2). Wird auf die Umgebung gewartet, so müssen nach obigen Regeln alle Umgebungsereignisse implementiert werden, womit die entsprechenden Transitionen alle in der Strategie sind (siehe Bedingung **g2**, ebenda).

3.3.4 Erzeugung neuer Kandidaten

Dieser Abschnitt beschreibt zunächst die eigentliche Erzeugung des neuen Kandidaten. Anschließend wird kurz erläutert, wie geprüft wird, ob der neu erzeugte Kandidat bereits existiert, also ein Duplikat ist.

Erzeugung eines neuen Kandidaten auf Basis des aktuellen Kandidaten

Abbildung 3.10 zeigt ein Aktivitätsdiagramm für die Erzeugung neuer Kandidaten. Als Basis dient der aktuelle Kandidat. Eine zusätzliche Eingabe sind die unimplementierten Zustände des aktuellen Kandidaten als Gegenbeispiel dafür, dass dieser Kandidat eine Lösung ist. Die dargestellten Schritte werden nachfolgend erläutert.

Wenn der Algorithmus einen oder mehrere unimplementierte SG-Zustände für den aktuellen Kandidaten (im Folgenden AK genannt) als Gegenbeispiel ermittelt hat, dann wählt er einen beliebigen Zustand s von diesen aus. Für s wird dann durch Kopieren von AK ein neuer Kandidat AK' erzeugt. Der Algorithmus versucht dann, Transitionen und Zustände zu AK' so hinzuzufügen, dass s in diesem neuen Kandidaten implementiert ist. Diese Erweiterungen hängen davon ab, ob s ein Zielzustand ist, oder nicht.

Der gewählte Zustand s ist ein Zielzustand: Wenn s ein Zielzustand ist, dann prüft der Algorithmus die folgenden beiden Bedingungen:

1. Es existiert kein anderer Kandidat AK'', der Nachfolger von AK im KG ist (d. h. es gibt keinen AK'' mit Kante von AK zu AK''), für den alle *unkontrollierbaren* Ereignisse in s implementiert sind.

2. Es gibt keinen Controller in AK, der in s sendet.

Bedingung 1 stellt sicher, dass der Algorithmus nicht versuchen wird, denselben Kandidaten mehrfach als Nachfolger von AK zu erzeugen. Bedingung 2 garantiert, dass

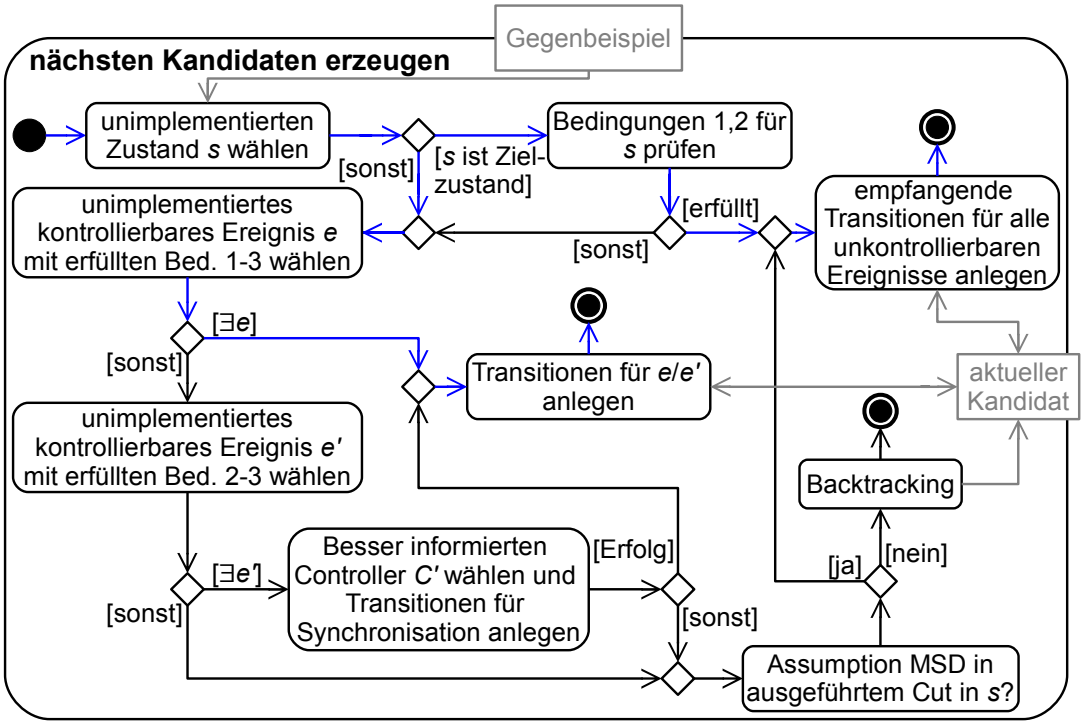


Abbildung 3.10: UML-Aktivitätsdiagramm zur Erzeugung neuer Kandidaten. Die Kanten für den Idealfall sind blau, der Objektfluss grau dargestellt.

Umgebungsereignisse nur in Zuständen betrachtet werden, in denen das System nicht bereits sendet. In diesen Zuständen können Umgebungsereignisse nach unserer Annahme, dass das System immer schneller ist als die Umgebung, nicht mehr auftreten.

Wenn beide Bedingungen erfüllt sind, dann fügt der Algorithmus empfangende Transitionen zu den Controllern von AK' so hinzu, dass alle unkontrollierbaren Ereignisse in s implementiert sind. Wenn mindestens eine der Bedingungen nicht erfüllt ist, dann behandelt der Algorithmus s genauso wie einen Zustand, der kein Zielzustand ist.

Der gewählte Zustand s ist *kein* Zielzustand: Wenn s kein Zielzustand ist, dann wählt der Algorithmus eines der unimplementierten *kontrollierbaren* Ereignisse e, das die folgenden Bedingungen erfüllt:

1. Es existiert kein anderer Kandidat AK'', der Nachfolger von AK im KG ist, in dem das Ereignis e im Zustand s implementiert ist.
2. Der Controller C des Objekts, das e sendet, sendet noch keine Nachricht in s.
3. Durch die Implementierung von e in s wird kein Zyklus von implementierten SG-Transitionen ohne Zielzustand geschlossen.

Bedingung 1 stellt sicher, dass kein weiterer Kandidat für dieselbe Kombination von Ereignis e und Zustand s als Nachfolger von AK erzeugt wird, wenn für diese Kombi-

nation bereits zuvor festgestellt wurde, dass sie nicht zu einer Lösung führt. Bedingung 2 stellt sicher, dass die erzeugten Controller deterministisch bleiben, d. h. sie senden in jedem Zustand nur maximal eine Nachricht. Bedingung 3 ist erforderlich, um die Gewinnbedingung **g1** (siehe Abschnitt 3.2.2) zu erfüllen.

Der Algorithmus prüft Bedingung 3 mittels einer Tiefensuche, ausgehend vom Folgezustand s' der SG-Transition für e in s. Diese Tiefensuche verwendet nur SG-Transitionen, die für AK implementiert sind, und endet an Zielzuständen. Daher erreicht die Tiefensuche nur dann s, wenn es einen Pfad implementierter SG-Transitionen von s' zu s gibt. Somit würde durch Implementieren der SG-Transition für e in s ein Zyklus implementierter SG-Transitionen *ohne Zielzustand* geschlossen. Dies würde **g1** verletzen.

Wenn ein Ereignis e existiert, das diese Bedingungen erfüllt, dann fügt der Algorithmus in AK' für e ausgehende sendende Transitionen zu allen Zuständen von C (Controller des Objekts, das e sendet) hinzu, die zu s korrespondieren. Ist auch das empfangende Objekt von e kontrollierbar, so wird dessen Controller in allen zu s korrespondierenden Zuständen eine empfangende Transition für e hinzugefügt. Somit ist durch die Erweiterungen in AK' das Ereignis e in s implementiert (siehe Abschnitt 3.3.3).

Falls es *kein* Ereignis e gibt, das alle Bedingungen erfüllt, dann sucht der Algorithmus stattdessen ein unimplementiertes kontrollierbares Ereignis e', das die Bedingungen 2 und 3 erfüllt, *aber nicht 1.* In diesem Fall wurde bereits ein Kandidat für Zustand s und Ereignis e' als Nachfolger von AK erzeugt, aber der Synthesealgorithmus hat ein Backtracking durchgeführt. Dann kann es sein, dass die Kombination von s und e' an sich nicht problematisch ist, der sendende Controller C aber nicht genügend Informationen hat, um den Zustand s von anderen SG-Zuständen s' zu unterscheiden, in denen ein Senden von e' die Spezifikation verletzt. Dies kommt immer dann infrage, wenn ein zu s korrespondierender Zustand von C neben s auch zu weiteren SG-Zuständen korrespondiert: Da C diese Zustände dann nicht unterscheiden kann, muss der Controller das Ereignis e' in allen von diesen senden, um e' in s senden zu können.

In diesen Fällen sucht der Algorithmus einen anderen Controller C', der zumindest einen zusätzlichen Zustand s' von s unterscheiden kann. Existiert mindestens ein solcher Controller C', so fügt der Algorithmus Transitionen zu den Controllern in AK' hinzu, sodass ein oder mehrere Controller C' Synchronisationsnachrichten an C senden. Diese erlauben C, s von s' zu unterscheiden. Nach Hinzufügen der Transitionen für die Synchronisation wird AK' wie oben beschrieben erweitert, um s in AK' zu implementieren – diesmal ohne e auch in s' zu senden.

Es kann passieren, dass auch nach erfolgter Synchronisation keine Synthese möglich ist und der Algorithmus erneut zu demselben AK zurückkehrt. Dann werden für dasselbe Ereignis e' zunächst alternative Synchronisationen für andere Mengen von Controllern C' erzeugt. Es wird erst dann ein anderes Ereignis e' gesucht, wenn alle möglichen Mengen an Controllern C' als Sender von Synchronisationsnachrichten versucht wurden, für die C möglichst viele Zustände von s unterscheiden kann.

Bei der konkreten Auswahl der Controller C' gibt es einen Trade-off: Werden zunächst nur möglichst kleine Mengen von Sendern gewählt, so kann es sein, dass sehr viele Fälle betrachtet werden müssen, wenn diese Synchronisationen nicht ausreichen. Dadurch kann der Berechnungsaufwand im Einzelfall sehr hoch sein. Andererseits werden so keine unnötigen Synchronisationsnachrichten eingefügt. Werden direkt so viele Sender be-

trachtet, dass der Empfänger den aktuellen SG-Zustand (möglichst) genau kennt, dann müssen bei Fehlschlag weniger Alternativen betrachtet werden – oder überhaupt keine, wenn C im Zustand nach der Synchronisation nur noch zu s korrespondiert. Allerdings kann es dann sein, dass nicht alle Sender tatsächlich eine Synchronisationsnachricht senden müssten.

Im Rahmen dieser Arbeit wurde als Kompromiss ein Greedy-Ansatz für die Auswahl der Synchronisations-Sender umgesetzt: Es wird zunächst mit einem einzigen Sender begonnen. Bei jedem Fehlschlag wird die Menge der Sender dann um einen erweitert – wobei jedoch nur Sender hinzugefügt werden, die die Menge korrespondierender Zustände für s in C nach der Synchronisation möglichst stark reduzieren.

Falls es kein Ereignis e' gibt, das die Bedingungen 2. und 3. erfüllt, wenn aber ein Assumption MSD in einem ausgeführten Cut in s ist, dann darf das System in s auf die Umgebung warten. In diesem Fall versucht der Algorithmus, alle Umgebungsereignisse in s zu implementieren, wie es oben für Zielzustände beschrieben wurde.

Backtracking In allen anderen Fällen verwirft der Algorithmus AK (und AK') und führt ein Backtracking durch. Er kehrt dann zum Vorgänger-Kandidaten von AK, AK_p zurück und versucht, nach obiger Beschreibung einen neuen Nachfolger für diesen zu finden. Dabei wird für AK_p derselbe SG-Zustand s ausgewählt wie bei der Konstruktion von AK. Dies stellt sicher, dass der Algorithmus nicht versuchen wird, andere SG-Zustände zu implementieren, wenn s nicht implementierbar ist. In diesem Fall kann durch Erweiterung von AK_p ohnehin keine Lösung gefunden werden und der Algorithmus muss erneut ein Backtracking durchführen.

Falls AK bereits mehrfach konstruiert wurde, so wurden die Duplikate zu einem Kandidaten zusammengefasst (siehe nächster Abschnitt). In diesem Fall hat AK mehrere Vorgänger, die jeweils über die eingehenden Kanten von AK im Kandidatengraph erreichbar sind. Es wird dann beim Backtracking zunächst ein beliebiger davon gewählt; die übrigen Vorgänger werden in einer Liste gespeichert. Aus dieser Liste wird immer dann ein Kandidat als AK gewählt, wenn ein Backtracking für den initialen Kandidaten durchgeführt werden müsste – der keine Vorgänger hat. Falls diese Liste jedoch leer ist und dennoch ein Backtracking des initialen Kandidaten erforderlich wäre, dann terminiert der Algorithmus ohne Lösung und meldet einen Fehlschlag der Synthese.

Entfernung von Duplikaten

Nach jeder Erweiterung des KG prüft der Algorithmus alle neuen Kandidaten und alle neuen Controllerzustände darauf, ob sie Duplikate von bereits existierenden Modellelementen sind. Duplikate werden jeweils zu einer einzigen Instanz verschmolzen. Dazu werden die eingehenden Kanten (bzw. Transitionen) der Duplikate auf diese einzige Instanz gerichtet. Kandidaten werden als Duplikate betrachtet, wann immer alle ihre Controller *identisch* sind. Wir betrachten Controller für das gleiche Systemobjekt als identisch, wenn die Mengen ihrer Zustände und Transitionen identisch sind.

Controllertransitionen sind identisch, wenn ihre Ausgangs- und Folgezustände, sowie ihr Ereignis, identisch sind. Controllerzustände sind identisch, wenn sie zu denselben

SG-Zuständen korrespondieren und wenn die Menge der anderen Controller, zu denen Synchronisationsnachrichten gesendet wurden, identisch ist. Die Berücksichtigung gesendeter Synchronisationsnachrichten ist notwendig, weil jedes Senden einer Synchronisationsnachricht zu einem neuen Controllerzustand führen muss, damit der Controller die Nachricht nicht erneut sendet. Andernfalls wäre jede sendende Transition einer Synchronisationsnachricht eine Selbsttransition, da die korrespondierenden Zustände nach dem Senden einer Synchronisationsnachricht beim Sender identisch bleiben.

3.3.5 Korrektheit

Im Folgenden wird informell für die Korrektheit des hier vorgestellten Algorithmus argumentiert. Dazu wird zunächst die Terminierung diskutiert. Dann wird für den Fall, dass eine Lösung zurückgegeben wird, argumentiert, dass diese eine korrekte Lösung des Problems der verteilten Synthese ist. Schließlich wird kurz erläutert, warum der Algorithmus vollständig ist, also immer dann eine Lösung liefert, wenn es eine gibt.

Terminierung

Der Synthesealgorithmus ist grundsätzlich eine Tiefensuche auf dem Kandidatengraphen und terminiert daher, wenn dieser Graph endlich ist. Der KG ist endlich, wenn die Anzahl möglicher Kandidaten für die gegebene Spezifikation endlich ist. Neue Kandidaten werden dem KG nur für neue Kombinationen von Controllern hinzugefügt. Deren maximale Anzahl wiederum hängt von der maximalen Anzahl von Controllerzuständen und Controllertransitionen ab. Ohne Berücksichtigung der zusätzlichen Synchronisationsnachrichten ist die maximale Anzahl von Controllerzuständen durch die Größe der Potenzmenge der SG-Zustände beschränkt, da neue Controllerzustände nur für neue Mengen von korrespondierenden SG-Zuständen erzeugt werden.

Für jede Synchronisationsnachricht muss im sendenden Controller auch dann ein neuer Controllerzustand angelegt werden, wenn für dieselbe Menge korrespondierender Zustände bereits Controllerzustände existieren. Dies ermöglicht dem Controller, den Zustand vor dem Senden vom Zustand danach zu unterscheiden. Synchronisationsnachrichten werden jedoch nur in Fällen hinzugefügt, in denen ein Controller zum Senden in einem SG-Zustand Informationen von anderen Controllern benötigt. Im schlimmsten Fall muss in einem SG-Zustand der sendende Controller von jedem anderen Controller eine Synchronisationsnachricht empfangen – was jeweils einen zusätzlichen Zustand im sendenden Controller erfordert. Selbst wenn das in jedem SG-Zustand erforderlich ist, ist die Anzahl der Controllerzustände also bei endlichem SG und endlicher Anzahl von Controllern ebenfalls endlich. Die Anzahl von Controllern entspricht direkt der Anzahl von Systemobjekten, die wir bei den hier betrachteten Systemen als endlich annehmen.

Die Anzahl möglicher Controllertransitionen hängt wiederum von der Anzahl der Controllerzustände und der Anzahl möglicher Ereignisse in der Spezifikation ab – die endlich sein muss, wenn der SG endlich ist. Insgesamt ist der KG also endlich, wenn der SG endlich ist – womit die Tiefensuche auf dem KG terminieren muss. Die Berechnung der korrespondierenden Zustände erfordert weitere Tiefensuchen, die aber ebenfalls auf Teilen des SG durchgeführt werden. Auch die Anzahl dieser Tiefensuchen ist bei endli-

chem SG endlich, da sie nur für neue Kandidaten durchgeführt werden. Insgesamt ist (bei endlichem SG) also garantiert, dass *der Algorithmus terminiert.*

Korrektheit der ermittelten Lösung

Wenn der Algorithmus erfolgreich terminiert, dann gibt er den zum jeweiligen Zeitpunkt aktuellen Kandidaten zurück. Der Algorithmus terminiert aber nur dann für einen Kandidaten, wenn es keine unimplementierten erreichbaren SG-Zustände für diesen gibt. Daher ist bei Terminierung jeder SG-Zustand entweder nicht in der Strategie, die durch den Kandidaten induziert wird, oder er ist implementiert.

Wenn ein SG-Zustand implementiert ist, dann bedeutet das (siehe Abschnitt 3.3.3), dass entweder der betreffende SG-Zustand ein Zielzustand ist und alle Umgebungsereignisse durch den Kandidaten implementiert werden oder der Kandidat mindestens eine Nachricht sendet und damit garantiert, dass der SG-Zustand kein Deadlock-Zustand ist. Die Strategie erfüllt daher die Gewinnbedingungen aus Abschnitt 3.2.2, bis auf den zweiten Teil von **g1**. Weiterhin prüft der Algorithmus auf Zyklen ohne Zielzustand und erzeugt keine Kandidaten, deren Controller Transitionen enthalten, welche zum Schließen dieser Zyklen führen würden. Damit verbleiben nur Zyklen implementierter Transitionen im System, die einen Zielzustand enthalten. Die Strategie erfüllt also auch den zweiten Teil der Gewinnbedingung **g1**.

Insgesamt entsprechen die lokalen Bedingungen für die Implementiertheit von SG-Zuständen also den globalen Bedingungen für eine Gewinnstrategie. Für den aktuellen Kandidaten muss bei Terminierung jeder SG-Zustand diese Bedingungen erfüllen. Insgesamt erfüllt die durch den zurückgegebenen Kandidaten induzierte Strategie also die Bedingungen **g1** und **g2** für eine Gewinnstrategie. *Daher ist der vom Algorithmus zurückgegebene Kandidat eine gültige Implementierung der gegebenen Spezifikation* – und damit eine Lösung für das Problem der verteilten Synthese für diese Spezifikation.

Vollständigkeit

Der Algorithmus erweitert jeden Lösungskandidaten so lange, bis dieser entweder eine Lösung ist, oder festgestellt wurde, dass er nicht mehr zu einer Lösung erweitert werden kann. Letzteres ist der Fall, wenn SG-Zustände, die einer Verletzung der Spezifikation entsprechen, erreichbar sind. Es ist auch dann der Fall, wenn ein SG-Zustand nicht implementiert werden kann, weil keine passenden Transitionen zur Erzeugung eines *neuen* Kandidaten hinzugefügt werden können – und die bereits auf diese Weise erzeugten Kandidaten schon als nicht zur Lösung erweiterbar identifiziert wurden.

Solange der Algorithmus keine Lösung gefunden hat, konstruiert er systematisch so lange alle möglichen Controllersystem-Kandidaten, bis er jeden als nicht zur Lösung erweiterbar identifiziert hat. Ist dies für das initiale Controllersystem der Fall, dann kann es keine Lösung geben. Der Algorithmus terminiert daher ohne Lösung, sobald er diesen Fall festgestellt hat. In allen anderen Fällen ist das initiale Controllersystem zu mindestens einer Lösung erweiterbar, die der Algorithmus zurückgibt, sofern er terminiert.

Insgesamt gilt also: *Wenn es eine Lösung gibt und der Algorithmus terminiert, dann gibt er diese Lösung zurück.*

3.3.6 Beispielausführung

Dieser Abschnitt erläutert die durch den Algorithmus ausgeführten Schritte anhand einer Anwendung auf das in Abschnitt 1.1 eingeführte Beispiel einer Produktionszelle. Dabei wird die Basisvariante der MSD-Spezifikation dieses Systems aus Abschnitt 3.1 als Eingabe verwendet. Die Abbildungen 3.11 und 3.12 zeigen für die wichtigsten Modelle des Algorithmus einige Zwischenstände, die bei der Synthese für das Beispiel entstehen. Im Einzelnen sind dies die Controller (grau hinterlegt) der Systemobjekte, welche zusammen das Controllersystem CS – also den aktuellen Kandidaten – bilden, der bisher explorierte Anteil des SG S (zwischen den beiden Controllern) und der Kandidatengraph (ganz rechts). Die weiteren Abbildungen zu Beispielausführungen des Algorithmus in dieser Arbeit verwenden eine ähnliche Darstellung.

Neben den bisher explorierten Zuständen und Transitionen zeigen die Abbildungen für den SG auch Nachrichtenereignisse, für die noch nicht der Folgezustand berechnet wurde. Für diese sind Transitionen eingezeichnet, obwohl diese (noch) nicht Teil des Graphen sind; die Folgezustände sind dann als gestrichelter Kreis mit einem Fragezeichen statt einer Zustandsnummer dargestellt. Die zum betreffenden Zeitpunkt erreichbaren unimplementierten SG-Zustände sind hier mit einem Ausrufezeichen gekennzeichnet. Zielzustände sind mit einem „G“ (für engl. ***g**oal state*) markiert.

Da das Strukturmodell für die Spezifikation in diesem Fall als Systemobjekte den PressController pC (das Steuergerät für die Presse) und den ArmController aC (das Steuergerät für den Roboterarm) definiert, erzeugt die Synthese für diese je einen Controller. Die Transitionen der dargestellten Controller sind jeweils mit dem Namen der gesendeten bzw. empfangenen Nachricht beschriftet. Auf den Namen folgt bei gesendeten Nachrichten ein „!“, bei empfangenen ein „?“. Zwischen den Controllerzuständen und den jeweils korrespondierenden SG-Zuständen sind Linien zur Visualisierung der Korrespondenzen eingezeichnet: Sind zwei Zustände mit einer (blauen) gepunkteten Linie verbunden, so sind diese korrespondierend.

Die Abbildungen 3.11 und 3.12 zeigen rechts jeweils den aktuellen Kandidatengraphen. Die Kandidaten sind in der Reihenfolge ihrer Erzeugung fortlaufend nummeriert. Für jede Erweiterung eines Kandidaten zu einem neuen Kandidaten ist die Kante mit dem SG-Zustand beschriftet, der dadurch neu implementiert wird. Die Beschriftung „env“ kennzeichnet die Implementierung aller unkontrollierbaren Ereignisse des Zustands, „sync“ kennzeichnet das Anlegen einer Synchronisation vor dem Implementieren eines Ereignisses. Sender und Empfänger der Synchronisationsnachricht werden in der Form `<Sendername>` $->$ `<Empfängername>` angegeben. Erweiterungen zur Implementierung kontrollierbarer Ereignisse sind mit dem Namen des betreffenden Ereignisses beschriftet. Im Kandidatengraph zeigen die Abbildungen auch diejenigen Kandidaten, die rückgängig gemacht wurden. Diese sind hier jeweils durch ein (rotes) „X“ markiert.

Im Folgenden werden zunächst einige Schritte des Standardablaufs der Synthese behandelt, die keine zusätzliche Kommunikation erfordern. Anschließend wird ein Schritt beschrieben, in dem eine Synchronisationsnachricht eingefügt werden muss, um fehlende Informationen an ein Objekt zu übermitteln, das diese benötigt. Dies führt für das Beispiel dann zur erfolgreichen Erzeugung der Controller. Schließlich wird ein alternativer Ablauf einiger Schritte des Verfahrens für dasselbe Beispiel beschrieben.

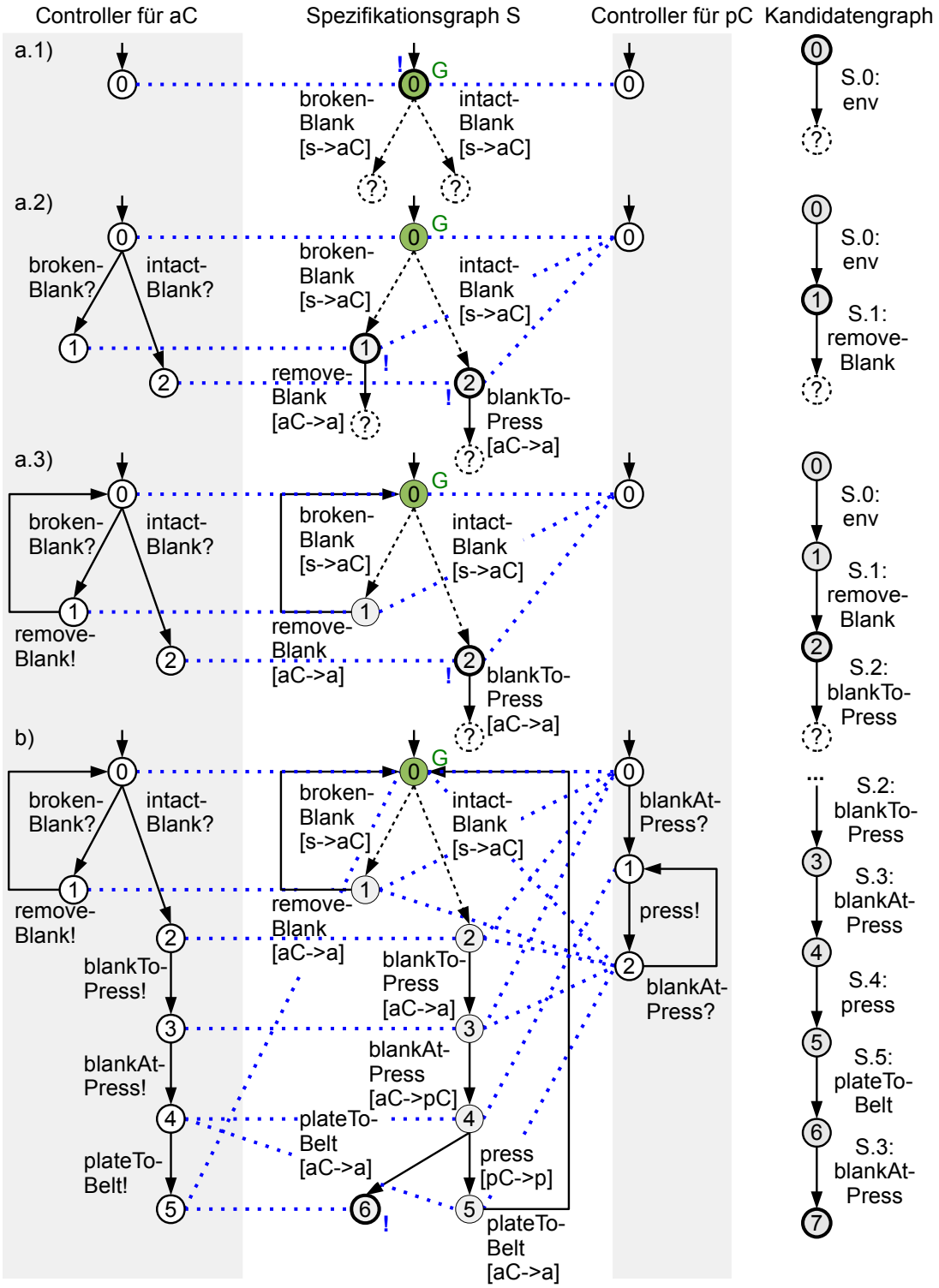


Abbildung 3.11: Beispieldurchlauf des Algorithmus für die Produktionszelle (Teil 1)

Standardablauf

Zwischenstand a.1: Der in Abbildung 3.11 **a.1** dargestellte Zwischenstand ist die Situation direkt nach Initialisierung der Modelle des Algorithmus. Der Algorithmus hat für beide Systemobjekte aC und pC jeweils einen Controller mit einem Startzustand angelegt. Beide Startzustände der Controller korrespondieren zu S.0, dem Startzustand des SG, der zunächst unimplementiert ist.

Da initial noch keine MSDs aktiv sein können, ist der Startzustand immer auch ein Zielzustand. Das System braucht also nichts zu senden, muss aber auf Nachrichten der Umgebung warten. Dies können hier `intactBlank` und `brokenBlank` sein, die jeweils zu einem neuen SG-Zustand führen. Beide Folgezustände wurden noch nicht exploriert.

Zwischenstand a.2: Um den Zielzustand S.0 zu implementieren, muss das Controllersystem alle (den Umgebungsannahmen nach) möglichen Umgebungsereignisse implementieren. Zwischenstand **a.2** zeigt die Situation nach Erzeugung eines neuen Kandidaten durch entsprechende Erweiterung des initialen Controllersystems.

Der neue Kandidat enthält empfangende Transitionen für `brokenBlank` und `intactBlank` in aC, wodurch S.0 implementiert wird. Dadurch werden aber auch die SG-Zustände S.1 und S.2 erreichbar. Der Controller für aC kann beide Ereignisse beobachten und „weiß“ daher, in welchem SG-Zustand das System nach einem der Ereignisse ist – also in S.1 bzw. S.2. Daher führen die neuen Controllertransitionen zu neuen Zuständen, die zu je einem SG-Zustand korrespondieren. Der Controller für pC andererseits kann diese Nachrichten nicht beobachten und kann daher S.1 und S.2 nicht voneinander oder von S.0 unterscheiden. Alle drei Zustände korrespondieren daher zu pC.0 (Zustand 0 des Controllers von pC). Der Algorithmus prüft für die erreichbaren Zustände S.0, S.1 und S.2, ob diese unimplementiert sind. Dies ist für S.1 und S.2 der Fall.

Zwischenstand a.3: Der Algorithmus wählt den unimplementierten Zustand S.1 aus. Da dieser kein Zielzustand ist, muss der Algorithmus ein Systemereignis auswählen, wofür hier nur das Senden von `removeBlank` infrage kommt. Im neuen zweiten Kandidaten wird dem Controller von aC daher eine neue Transition hinzugefügt, die diese Nachricht sendet. Dabei wird festgestellt, dass die entsprechende SG-Transition zu S.0 zurückführt, da nach diesem Ereignis dieselben aktiven MSDs in denselben Cuts sind.

Um mögliche zusätzliche Korrespondenzen des Folgezustands der neuen Transition im Controller von aC zu ermitteln, wird eine Tiefensuche von S.0 aus über nicht für aC beobachtbare SG-Transitionen ausgeführt. Diese terminiert jedoch sofort, da alle ausgehenden Transitionen in S.0 beobachtbar für aC sind. Der durch die neue Controllertransition erreichte Zustand korrespondiert also nur zu S.0. Im Controller aC existiert bereits der Zustand aC.0 mit genau dieser Zustandskorrespondenz, zu dem die Transition daher zurück führt. Durch die neue Transition in aC ist S.1 nun implementiert. Da alle Ereignisse in Zustand S.0 weiterhin implementiert sind, bleibt S.0 ebenfalls implementiert. Daher verbleibt nur S.2 als unimplementierter Zustand.

Zwischenstand b: In Abbildung 3.11 werden von Zwischenstand **a.3** bis **b** einige Schritte übersprungen. In **b** ist der Zustand direkt nach der Erzeugung eines Kandi-

daten dargestellt, der eine neue Transition enthält, welche den zweiten Zyklus im SG zurück zum Startzustand (und Zielzustand) S.0 schließt.

In den zwischenzeitlich erzeugten Kandidaten hat der Algorithmus – wie bereits in Schritt **a.3** – den Controller aC um Transitionen erweitert, die Nachrichten senden. Während die Adressaten der ersten und der letzten Nachricht Umgebungsobjekte sind, wird die zweite Nachricht, also `blankAtPress`, an pC gesendet. Daher fügt der Algorithmus bei Erzeugung des entsprechenden Kandidaten auch dem Controller für pC eine Transition für `blankAtPress` hinzu, aber diesmal eine empfangende. Der Folgezustand pC.1 der neu erzeugten Transition korrespondiert nur zu S.4, d. h. pC kann diesen Zustand von den übrigen SG-Zuständen unterscheiden. Der Controller für pC darf in diesem Zustand die Nachricht `press` senden, was S.4 implementieren würde. Alternativ kann der Controller für aC dazu in S.4 die Nachricht `plateToBelt` senden. Der Algorithmus entscheidet sich hier für die erste Variante, also dafür, dass pC `press` sendet.

Mangels Alternativen wird anschließend dennoch `plateToBelt` durch aC gesendet, um S.5 zu implementieren. Dadurch wird in der Spezifikation der Zyklus zum Startzustand S.0 geschlossen. Da aC.4 jedoch sowohl zu S.4 als auch zu S.5 korrespondiert, sendet aC die Nachricht `plateToBelt` in beiden Zuständen. Somit wird auch S.6 erreichbar. Der Zustand im Controller von aC nach dem Senden von `plateToBelt` korrespondiert somit sowohl zu S.0 (durch Senden in S.5) als auch zu S.6 (durch Senden in S.4).

Die Nachricht `plateToBelt` und die darauffolgenden Nachrichten vor `blankAtPress` sind jedoch durch pC nicht beobachtbar. Der Zyklus zu Zustand S.0 wird geschlossen während sich pC in Zustand pC.2 befindet. Dieser Controllerzustand ist zunächst nur zu S.5 korrespondierend. Durch das Schließen des Zyklus im SG kann pC.2 jedoch zu neuen SG-Zuständen korrespondierend werden (vgl. Definition von `corr` in Abschnitt 3.3.3).

Um die neuen korrespondierenden Zustände für pC.2 zu ermitteln, wird eine Tiefensuche vom erneut erreichten Zustand S.0 aus über von pC nicht beobachtbare Transitionen durchgeführt. Die als Ergebnis ermittelten Korrespondenzen sind in Abbildung 3.11 **b** eingezeichnet und enthalten unter anderem S.3. Dadurch wird S.3 unimplementiert, weil die Nachricht `blankAtPress` nun auch in pC.2 empfangen werden muss, damit S.3 implementiert ist. Aus diesem Grund wird der Kandidat 7 erzeugt, der eine neue ausgehende Transition in pC.2 enthält, die im Controller von pC einen Zyklus schließt.

Der Zustand S.3 ist damit wieder implementiert; S.6 bleibt jedoch unimplementiert.

Einfügen einer Synchronisation

Zwischenstand c.1: Der neue Zustand S.6 ist aufgrund einer Safety Violation in einem Requirement MSD ein Deadlock-Zustand im SG; d. h. kein Ereignis führt mehr zu einer Änderung des Zustands. Da die Assumption MSDs in diesem Zustand nicht verletzt oder in einem ausgeführten Cut sind, kann der Zustand nicht implementiert werden.

Um die Erreichbarkeit von S.6 zu verhindern, kehrt der Algorithmus daher schrittweise zu früheren Kandidaten zurück bis die sendende Transition für `plateToBelt` in Kandidat 5 nicht mehr implementiert ist. Abbildung 3.12 **c.1** zeigt den Zustand der Modelle des Algorithmus *nach* dem Zurückkehren zu diesem Kandidaten, d. h. alle nach diesem Kandidaten hinzugefügten Elemente wurden bereits wieder entfernt. Die mit einem „X“ markierten Kandidaten im KG wurden verworfen.

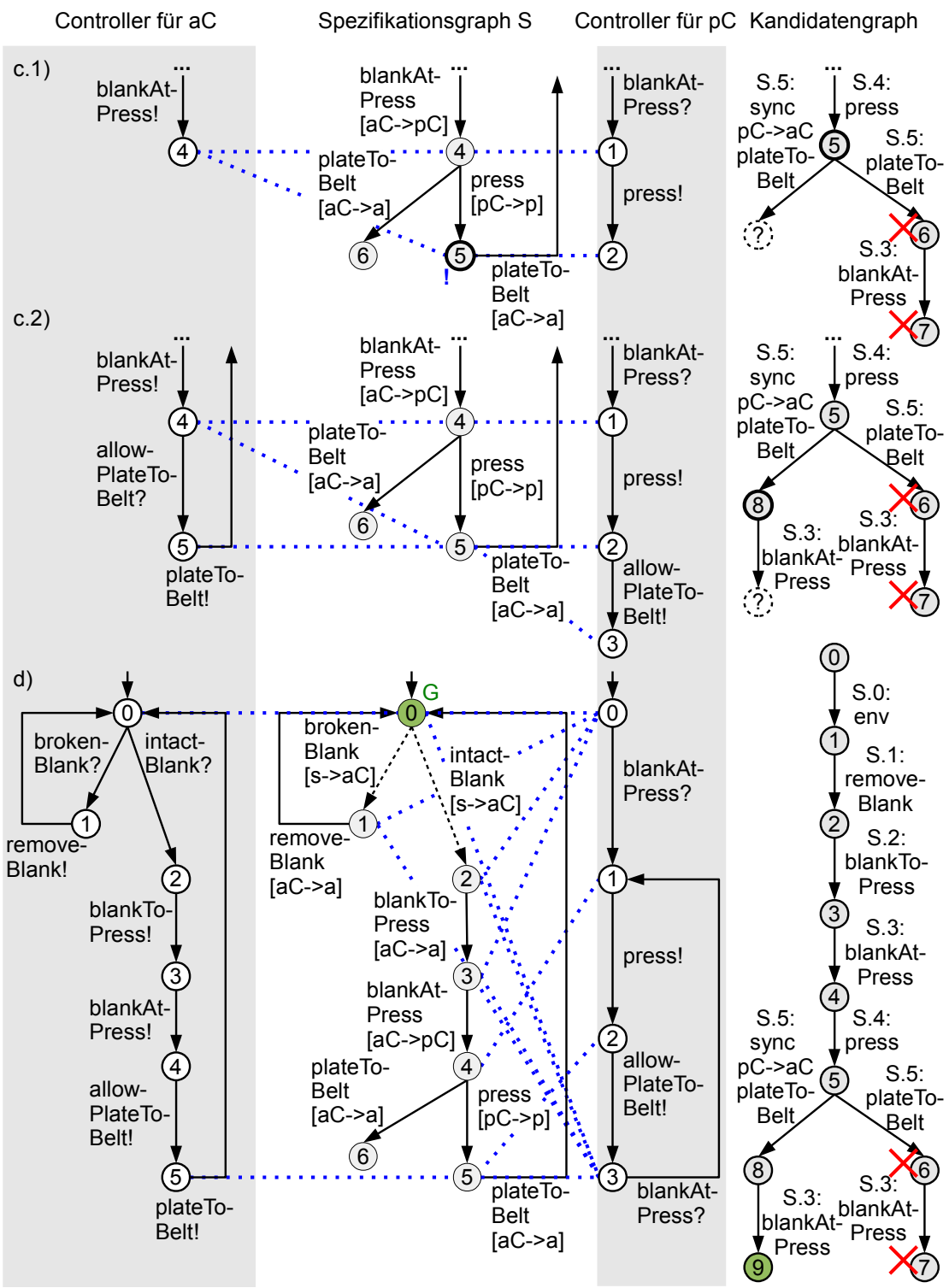


Abbildung 3.12: Beispieldurchlauf des Algorithmus für die Produktionszelle (Teil 2)

Zwischenstand c.2: Der Algorithmus muss nun Zustand S.5 erneut implementieren und letztendlich von diesem aus sicherstellen, dass ein Zielzustand erreicht wird. Aufgrund der entsprechenden ausgehenden Kante für Kandidat 5 stellt der Algorithmus fest, dass das Senden der Nachricht `plateToBelt` bereits erfolglos versucht wurde, während andererseits der SG selbst generell ein Senden von `plateToBelt` in S.5 erlaubt.

Der Grund dafür, dass aC `plateToBelt` nicht senden darf, ist, dass dieser Controller S.5 nicht von S.4 unterscheiden kann. Diese Unterscheidung ist jedoch notwendig, um ein erneutes Erreichen des Deadlock-Zustands S.6 auszuschließen. Der Controller von pC kann S.4 und S.5 aber durchaus unterscheiden, da der Zustand pC.2 zu S.5 korrespondiert, nicht jedoch zu S.4.

Der Algorithmus erzeugt daher einen Kandidaten, der Transitionen für eine entsprechende Synchronisationsnachricht `allowPlateToBelt` von pC zu aC in den beiden Controllern enthält. Die Nachricht `allowPlateToBelt` wurde nicht als Teil der Spezifikation definiert; sie dient lediglich dazu, aC das Erreichen von S.5 mitzuteilen und somit dazu, dass `plateToBelt` nun gesendet werden darf (und irgendwann gesendet werden muss). Nach Empfang der Synchronisationsnachricht kann aC nun in aC.5 (anders als zuvor in aC.4) S.5 von S.4 unterscheiden und darf daher `plateToBelt` senden.

Erfolgreiche Vervollständigung der Controller

Nach Einfügen der Synchronisation in Kandidat 8 muss der Algorithmus in S.4 nicht mehr die Nachricht `plateToBelt` implementieren, da sie nicht mehr in einem korrespondierenden Zustand von S.4 gesendet wird. Daher wird der problematische Zustand S.6 nicht erneut erreichbar.

Der Algorithmus muss also nur noch `blankAtPress` in S.3 implementieren, wie es für Zwischenstand **b** in Abbildung 3.11 diskutiert wurde. Somit sind schließlich alle erreichbaren SG-Zustände implementiert und der Algorithmus terminiert erfolgreich. Abbildung 3.12 **d** zeigt den Endzustand der Modelle nach Terminierung des Algorithmus. Aus Gründen der Lesbarkeit werden hier die Korrespondenzbeziehungen nur für einige der Controllerzustände gezeigt.

Alternativer Ablauf

Der Algorithmus hat in einem gegebenen SG-Zustand oft die Wahl, welche von mehreren ausgehenden Transitionen er implementiert. Bei dieser Auswahl können Heuristiken zum Einsatz kommen, die möglichst diejenigen Transitionen wählen, welche am wahrscheinlichsten zu einer Lösung führen werden. In dieser Arbeit wurde eine Variante umgesetzt, die bei mehreren kontrollierbaren Transitionen diejenigen vermeidet, die sofort eine Verletzung der Anforderungen bewirken. Sofern die Heuristik mehrere Transitionen auswählt, so ist die endgültige Auswahl aus dieser Menge beliebig. Von dieser Auswahl hängt jedoch ab, welchen Anteil des SG (wie viele „Sackgassen“) der Algorithmus durchsucht und wie viele Schritte er benötigt, bevor er terminiert.

Beispielsweise kann ein Zielzustand mit mehreren unkontrollierbaren ausgehenden Transitionen bereits dann nicht mehr implementiert werden, wenn auch nur eine davon zur Verletzung der Spezifikation führt. Der Algorithmus muss dann nur eine proble-

matische Transition identifizieren. Er kann aber durch ungünstige Auswahl auch alle anderen Transitionen – und das dadurch erreichbare Verhalten – zuerst betrachten und damit deutlich mehr Verhalten als nötig untersuchen. Dabei kann jede dieser Transitionen zur Exploration großer Bereiche des SG durch den Algorithmus führen. Die Anzahl von Schritten, die der Algorithmus durchführen muss, und daher auch dessen Laufzeit, kann somit stark von dessen nichtdeterministischen Entscheidungen abhängen.

Abbildung 3.13 zeigt Zwischenstände eines alternativen Ablaufs des Algorithmus, bei dem dieser die im Zusammenhang mit Zwischenstand **b** (s.o.) erwähnte nichtdeterministische Entscheidung anders trifft, d. h. aC sendet direkt `plateToBelt` anstatt dass pC `press` sendet. Die Zwischenstände **e.1** und **e.2** entsprechen direkt den Zwischenzuständen **c.1** bzw. **c.2**. Zwischenstand **e.1** zeigt also den Zustand direkt nach Entfernen der sendenden Transition für `plateToBelt` aus pC durch Backtracking.

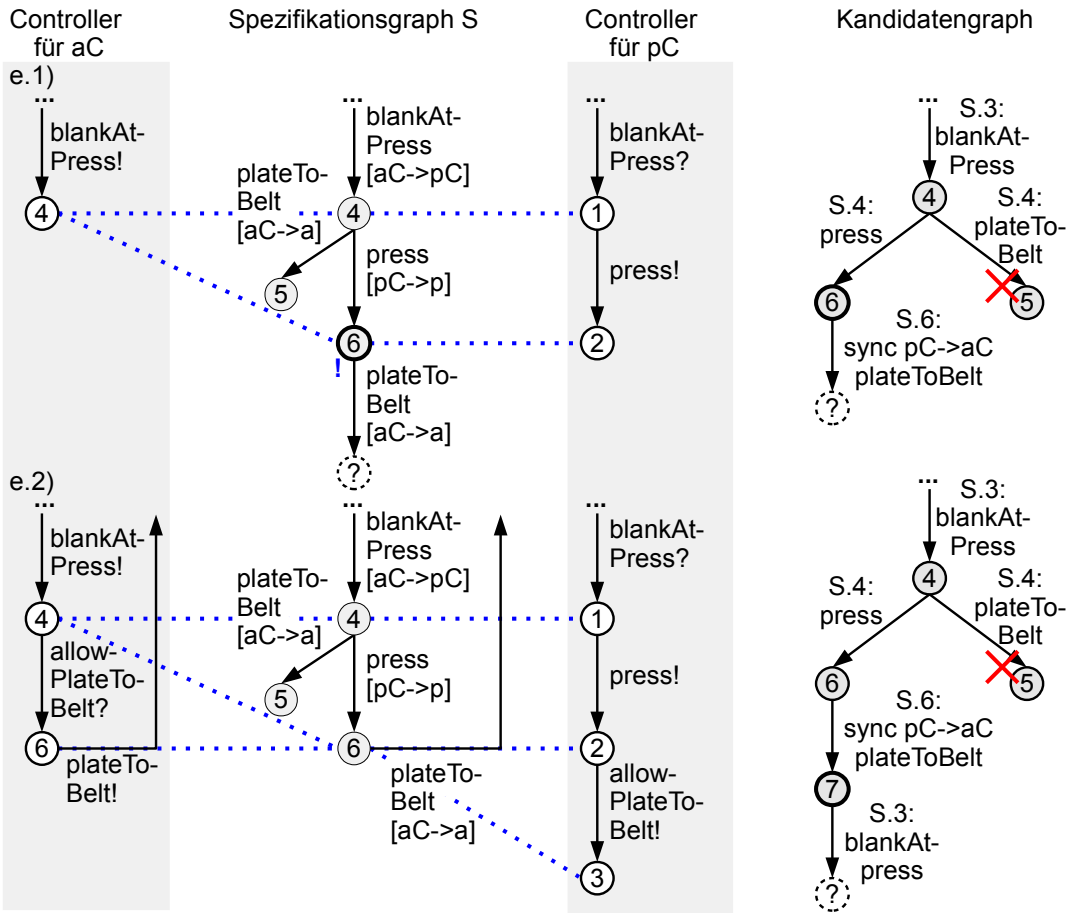


Abbildung 3.13: Alternativer Ablauf für **c.1** und **c.2** in Abbildung 3.12

Der Hauptunterschied des alternativen Ablaufs zum oben beschriebenen ist, dass nur ein Kandidat verworfen werden muss. Außerdem werden hier nie in demselben SG-Zustand zwei Nachrichten von Systemobjekten gesendet, wie es bei der oben beschriebenen Ausführung des Algorithmus in S.4 der Fall war. Da die SG-Zustände in der Reihenfolge

ihrer erstmaligen Explorierung nummeriert sind, sind hier die Nummern der Zustände S.5 und S.6 vertauscht, d. h. S.5 ist nun der Deadlock-Zustand. Wie zuvor in **c.2** werden auch in **e.2** Transitionen für eine Synchronisationsnachricht eingefügt. Der weitere Ablauf ist dann ebenfalls identisch zu dem oben beschriebenen. Insgesamt benötigt der alternative Ablauf einen Schritt weniger als der erste.

3.3.7 Spezialfall kombinierter Synchronisationen

Im in Abschnitt 3.3.6 beschriebenen Beispiel genügt der Austausch einer Synchronisationsnachricht für eine erfolgreiche Synthese. Es gibt jedoch Fälle, in denen ein Objekt vor dem Senden einer Nachricht Informationen von nicht nur einem, sondern mehreren anderen Objekten benötigt. Dieser Abschnitt gibt zunächst ein Beispiel für solch einen Fall und erläutert dann dessen Lösung durch eine *Kombination* von Synchronisationen.

Erweitertes Beispiel

Dieser Abschnitt erweitert die in Abschnitt 3.1 beschriebene MSD-Spezifikation so, dass statt einer nun zwei Synchronisationsnachrichten nacheinander ausgetauscht werden müssen. Das Beispiel wird dazu so erweitert, dass das Ablageförderband nun meldet, wenn es genug Platz für neue Metallplatten bietet. Es sendet dazu eine Nachricht an ein eigenes Steuergerät. Erst danach darf der Arm durch die Nachricht `plateToBelt` angewiesen werden, eine Metallplatte von der Presse zum Ablageförderband zu bewegen.

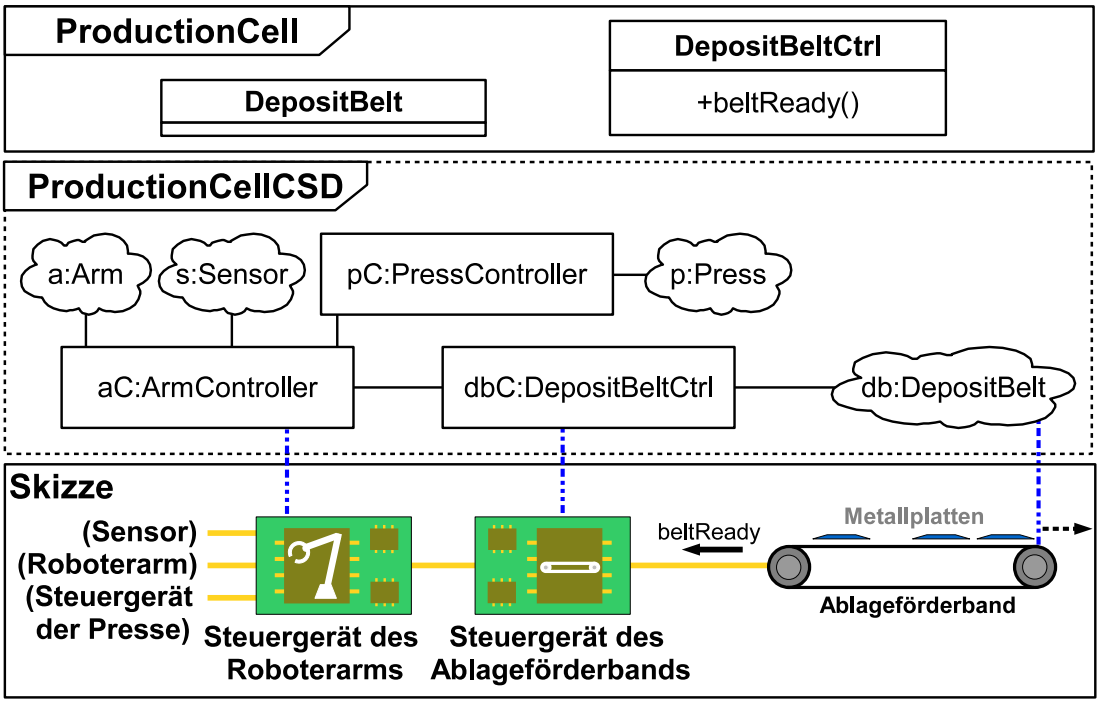
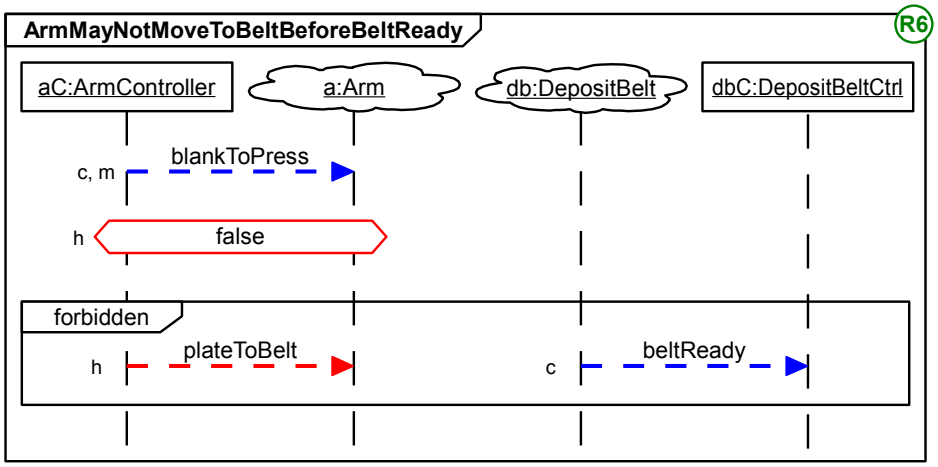


Abbildung 3.14: Neue Struktur zur Erläuterung kombinierter Synchronisationen

Abbildung 3.14 zeigt die Erweiterung der Strukturdefinition gegenüber der in Abbildung 3.1 gezeigten: Wie in der Skizze (unten) informell dargestellt ist, kann das Ablageförderband seinem Steuergerät durch Senden der Nachricht `beltReady` mitteilen, dass es zum Annehmen einer weiteren Metallplatte bereit ist. Zusätzlich existiert eine Kommunikationsverbindung zwischen diesem neuen Steuergerät und dem Steuergerät des Roboterarms.

Das Kompositionsstrukturdiagramm (Mitte) und das Klassendiagramm (oben) formalisieren die beschriebene Struktur. Zusätzlich definieren diese, dass das Steuergerät dbC der neuen Klasse DepositBeltCtrl (für engl. *deposit belt controller*) ein Systemobjekt ist, während das Ablageförderband db (für engl. *deposit belt*) der ebenfalls neuen Klasse DepositBelt als Umgebungsobjekt definiert ist. Als einzige neue Nachricht wird `beltReady` durch die Definition der entsprechenden Operation für DepositBeltCtrl eingeführt.

Abbildung 3.15 zeigt das neue Requirement MSD für Anforderung **R6**, das zu den Requirement MSDs in den Abbildungen 3.2 und 3.3 hinzukommt. Das neue MSD formalisiert, dass eine explizite Meldung durch das Ablageförderband erfolgen muss. Diese Anforderung wird ähnlich zu **R5** durch eine `false`-Bedingung ausgedrückt, nach deren Erreichen die Spezifikation nur durch Terminierung des aktiven MSDs mittels einer verbotenen Nachricht erfüllt werden kann. Unter dem MSD ist Anforderung **R6** zusätzlich in natürlichsprachlichem Text aufgeführt.



„Es darf *nicht* passieren, dass das Steuergerät des Arms den Arm anweist, einen Rohling zur Presse zu bewegen und anschließend die fertige Metallplatte zum Ablageförderband zu bewegen, *ohne dass* das Ablageförderband zwischenzeitlich bestätigt hat, dass es bereit ist.“

Abbildung 3.15: Erweiterung der MSD-Spezifikation in den Abbildungen 3.2, 3.3 und 3.4 zur Erläuterung kombinierter Synchronisationen

Behandlung des Beispielproblems

Die Ausführung des Algorithmus für die erweiterte MSD-Spezifikation führt zu einer Situation, in der zunächst kein einzelnes Objekt den genauen SG-Zustand kennt. Der

Controller für aC benötigt diese Information jedoch zum Senden einer Nachricht. Abbildung 3.16 zeigt mit **f.1** und **f.2** Zwischenstände der Controller, die ähnlich zu den in Abbildung 3.12 gezeigten Zwischenständen **c.1** und **c.2** sind. Neben dem zusätzlichen Controller für dbC ist der einzige Unterschied, dass hier in **f.2** eine zusätzliche Synchronisationsnachricht erforderlich ist.

Zwischenstand **f.1** zeigt zunächst die problematische Situation, die bei Ausführung der Synthese für die erweiterte MSD-Spezifikation entsteht. Zwischenstand **f.2** zeigt die Situation nach der Lösung des Problems durch Einfügen zweier Synchronisationen.

Zwischenstand **f.1** zeigt die Situation nach dem erfolglosem Versuch des Sendens von `plateToBelt` und anschließendem Backtracking durch den Algorithmus. In dieser Situation kann der ArmController aC S.8 weder von S.6 noch von S.7 unterscheiden, da er beide Nachrichten nicht beobachten kann. Die Steuergeräte pC und dbC können jedoch jeweils nicht die Nachricht beobachten, die das andere sendet. Daher reicht es hier – anders als in **c.1** (Abbildung 3.12) – nicht aus, pC nach Senden von `press` eine Synchronisationsnachricht an aC senden zu lassen. Der neue Controller für dbC, andererseits, empfängt `beltReady` von db und kann S.8 daher von S.7 unterscheiden, jedoch nicht von S.6. Daher ist es hier nötig, dass beide Controller – sowohl der von pC, als auch der von dbC – Synchronisationsnachrichten an aC senden, bevor dieser `plateToBelt` sendet.

Die dafür vom Algorithmus angelegten Transitionen sind in Zwischenstand **f.2** dargestellt. Durch die erste Synchronisationsnachricht hat aC in Zustand aC.5 zunächst dieselben Informationen wie der Sender dieser Nachricht, also dbC, in Zustand dbC.1. Erst durch die zweite Synchronisationsnachricht von pC „weiß“ aC dass der SG-Zustand S.8 erreicht ist, da dies der einzige zu aC.6 korrespondierende Zustand ist. Dies ergibt sich nach der Definition von *corr* in Abschnitt 3.3.3 durch eine Schnittmengenbildung der Mengen der korrespondierenden Zustände von aC.5 (identisch zu der von dbC.1) und pC.2. Damit darf der Controller für aC, der sich nach Empfang der Synchronisationsnachrichten in aC.6 befindet, anschließend die Nachricht `plateToBelt` senden. Die sendende Transition für `plateToBelt` in aC.6 wurde hier noch nicht erzeugt.

Um den hier beschriebenen Fall zu behandeln, ist keine Anpassung des Algorithmus aus Abschnitt 3.3 erforderlich. Insbesondere wurde in Abschnitt 3.3.4 bereits beschrieben, in welchen Fällen mehrere Synchronisationen statt nur einer eingefügt werden.

3.4 Berücksichtigung eingeschränkter Kommunikation

In den vorhergehenden Abschnitten wurde vereinfachend angenommen, dass die Systemobjekte uneingeschränkt miteinander kommunizieren können. In realen Systemen ist dies jedoch oft nicht der Fall. Beispielsweise skalieren Bussysteme nicht für beliebig viele Teilnehmer, direkte Kabelverbindungen können zu teuer sein, und kabellose Kommunikation hat eine beschränkte Reichweite.

Werden bestehende Einschränkungen der Kommunikation bei der Synthese nicht beachtet, so funktioniert die durch diese erzeugte Implementierung möglicherweise nicht korrekt in der Praxis. Dieser Fall kann auch dann auftreten, wenn die gegebene Spezifikation eine alternative Implementierung zulässt, die nur die definierten Verbindungen nutzt. Ohne Berücksichtigung der Architektur kann die Synthese eine solche korrek-

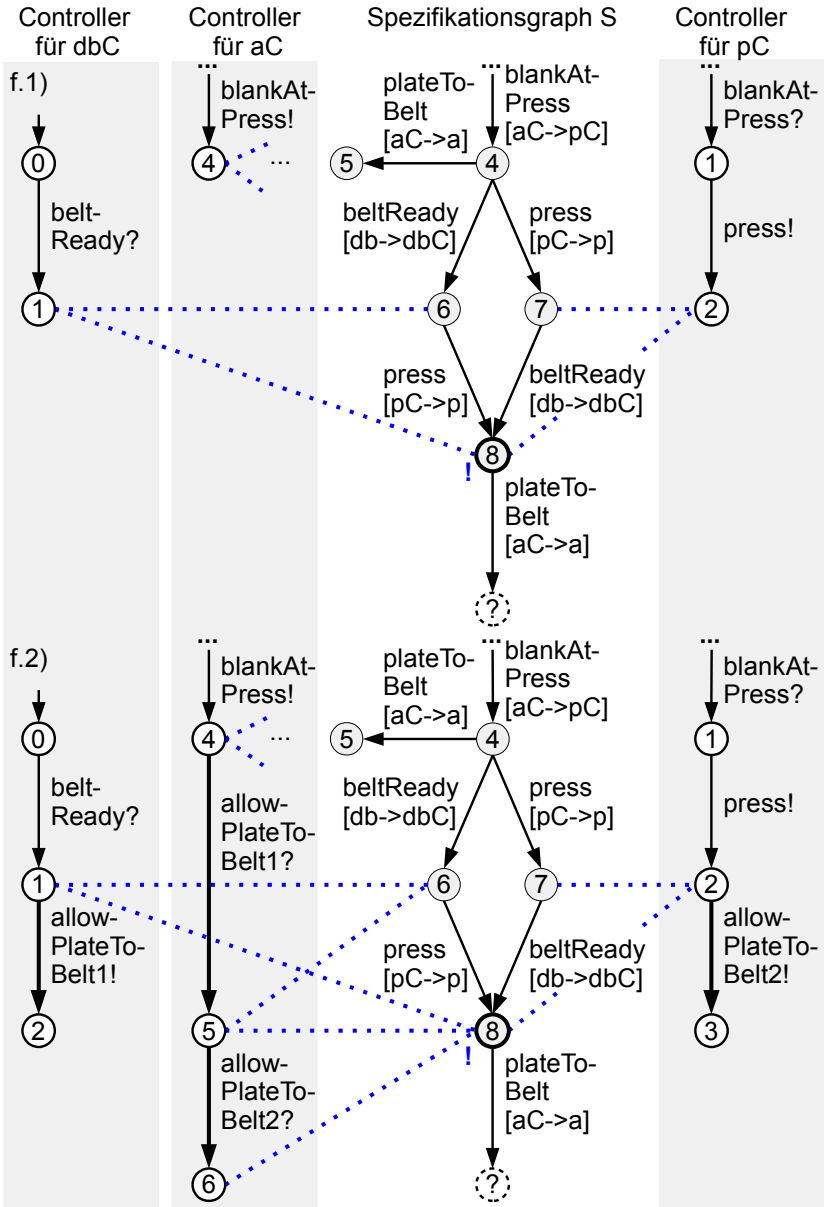


Abbildung 3.16: Die bei Ausführung des Algorithmus für die erweiterte Spezifikation gegenüber Abbildung 3.12 veränderten Zwischenstände

te Implementierung nur zufällig finden und nicht von inkorrekten Implementierungen unterscheiden. Daher wird hier erläutert, wie der in Abschnitt 3.3 beschriebene Syntheseansatz angepasst werden muss, um die Architektur explizit zu berücksichtigen – und dadurch garantiert eine implementierbare Lösung zu erzeugen.

In Abschnitt 3.4.1 wird die in Abschnitt 3.1 eingeführte MSD-Spezifikation erweitert, um ein Beispiel für architekturbezogene Einschränkungen der Kommunikation zu gewinnen. An diesem Beispiel wird in Abschnitt 3.4.2 gezeigt, wie die Einschränkungen bei Ausführung des Syntheseralgorithmus berücksichtigt werden können. Abschließend wird in Abschnitt 3.4.3 die dafür erforderliche Anpassung des Algorithmus diskutiert.

3.4.1 Erweitertes Beispiel

Dieser Abschnitt erweitert die in Abschnitt 3.1 beschriebene MSD-Spezifikation um Einschränkungen der möglichen Kommunikationsverbindungen. Insbesondere wird durch diese die im Beispiel erforderliche Synchronisation auf direktem Wege unmöglich. In der neuen Variante des Beispiels gibt es jedoch ein neues Systemobjekt, mittels dessen diese Synchronisation indirekt in zwei Schritten durchgeführt werden kann. Die auf das Kommunikationsverhalten bezogenen Anforderungen und Annahmen des ursprünglichen Beispiels bleiben in der erweiterten Variante unverändert.

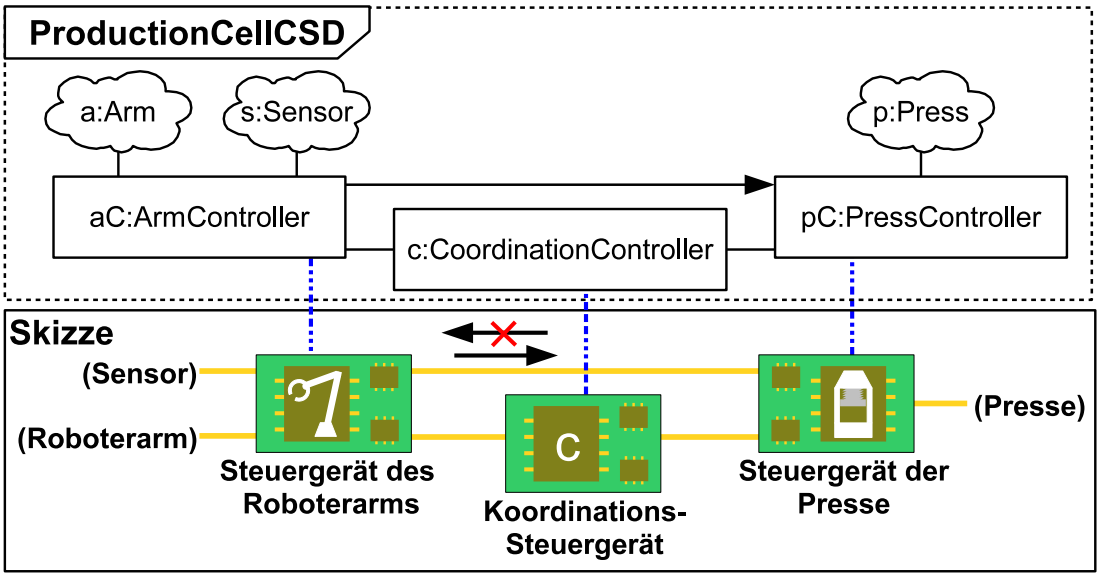


Abbildung 3.17: Gegenüber Abbildung 3.1 geänderte Struktur des erweiterten Beispiels für Synchronisation mit Einschränkungen der Kommunikation

Abbildung 3.17 zeigt die Änderungen der Strukturdefinition für das erweiterte Beispiel gegenüber der in Abbildung 3.1 gezeigten Struktur des ursprünglichen Beispiels. Die Skizze (unten) zeigt bereits, dass die Kommunikation vom Steuergerät der Presse zum Steuergerät des Arms nicht möglich ist. Die in der MSD-Spezifikation explizit definierte Kommunikation in die Gegenrichtung wird vom neuen Strukturmodell jedoch

weiterhin unterstützt. Die neue Struktur enthält ein zusätzliches Steuergerät, das Koordinationssteuergerät, das in beide Richtungen sowohl mit dem Arm-Steuergerät, als auch mit dem Pressen-Steuergerät kommunizieren kann. Für dieses neue Systemobjekt werden keine zusätzlichen Anforderungen definiert.

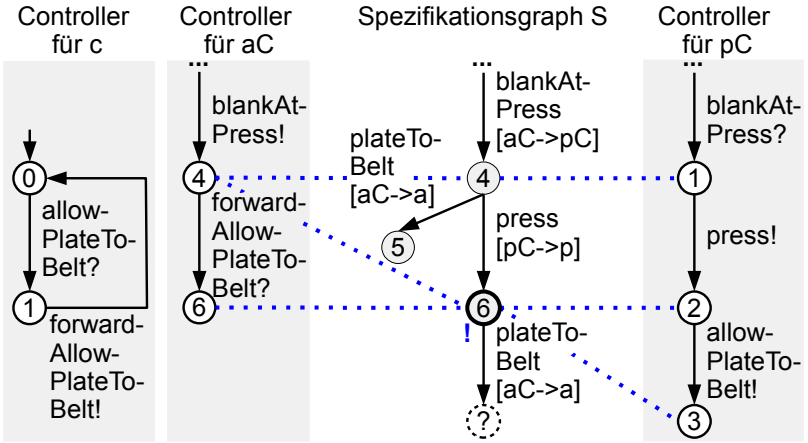


Abbildung 3.18: Beispiel für Synchronisationen unter Berücksichtigung von Einschränkungen der Kommunikation durch die Architektur

Die Skizze wird im Kompositionsstrukturdiagramm (oben) formalisiert: Ein ungerichteter Konnektor zwischen zwei Objekten bedeutet, dass diese miteinander in beide Richtungen kommunizieren können. Ein gerichteter Konnektor erlaubt nur das Senden von Nachrichten in Pfeilrichtung. Im Beispiel wird also definiert, dass die Kommunikation von pC zu aC nicht möglich ist, während aC an pC senden darf. Gibt es keinen Konnektor zwischen zwei Objekten, so ist keine Kommunikation möglich. Das Koordinationssteuergerät c darf mit aC und pC kommunizieren – und diese mit ihm. Als einzige neue Klasse kommt CoordinationController hinzu, die keine Operationen definiert.

3.4.2 Behandlung des Beispielproblems

Abbildung 3.18 zeigt einen Zwischenstand der Ausführung des Algorithmus auf das erweiterte Beispiel. Dieser illustriert, wie Synchronisationen auch dann durchgeführt werden können, wenn keine direkte, aber eine indirekte Kommunikation vom Sender der Nachricht zum Empfänger möglich ist. In der dargestellten Situation werden Synchronisationsnachrichten ausgetauscht, um dem Steuergerät pC die aC bekannte Information über den aktuellen globalen Zustand des Systems zukommen zu lassen.

Vor Einfügen der Synchronisationen ist die Situation identisch zu derjenigen, die in Abbildung 3.12 **c1** für die Basisvariante des Beispiels gezeigt wird, d. h. pC kann zwei Zustände des Spezifikationsgraphen (SG) nicht unterscheiden. Der Algorithmus erkennt zunächst aufgrund der Annotationen am Strukturmodell, dass keine direkte Kommunikation von pC zu aC möglich ist. Durch eine Tiefensuche im Kompositionsstrukturdiagramm (CSD, engl. *composite structure diagram*) stellt er jedoch fest, dass pC über das neue Systemobjekt c Nachrichten an aC schicken kann.

Der Algorithmus legt die erforderlichen Synchronisationen an, indem er den Kandidaten so um neue Transitionen erweitert, dass c die Synchronisationsnachricht von pC an aC weiterleitet. Dazu ist zunächst c Empfänger einer Synchronisationsnachricht von pC und hat in c.1 damit dieselben korrespondierenden Zustände wie pC.2, also nur SG-Zustand S.6 (in der Abbildung nicht eingezeichnet). In c.1 sendet c dann eine Synchronisationsnachricht an aC, der dann in aC.6 ebenfalls nur zu S.6 korrespondiert. Nun darf aC `plateToBelt` senden. Die entsprechende Transition wurde hier noch nicht erzeugt.

3.4.3 Erweiterung des Algorithmus

Um den Synthesealgorithmus so anzupassen, dass er architekturbedingte Kommunikationseinschränkungen unterstützt, muss verhindert werden, dass dieser Kommunikation zwischen Objekten anlegt, die diesen Einschränkungen nach nicht möglich ist. Dies betrifft zum einen zusätzlich eingefügte Synchronisationen. Zum anderen betrifft dies auch Kommunikation, die in der Spezifikation definiert ist. Zusätzlich muss berücksichtigt werden, dass eine Implementierung von Umgebungsereignissen nicht erforderlich ist, wenn diese architekturbedingt nie auftreten können.

Im Folgenden werden die nötigen Anpassungen des Algorithmus erläutert. Dabei wird zunächst das Einfügen von Synchronisationsnachrichten und dann das Implementieren von Nachrichten der Spezifikation betrachtet.

Anpassung des Einfügens von Synchronisationen

Um ausschließlich Synchronisationen zu erzeugen, die die Kommunikationseinschränkungen einhalten, muss der Algorithmus beim Erzeugen eines neuen Kandidaten bei jedem Einfügen einer Synchronisation das Strukturmodell berücksichtigen. Konkret liefert das Strukturmodell für ein gegebenes Paar von Sender- und Empfängerobjekt die Information, ob der Sender an den Empfänger senden darf.

Zudem muss, sofern keine direkte Kommunikation möglich ist, indirekte Kommunikation über andere Systemobjekte als Alternative betrachtet werden. Objekte können indirekt miteinander kommunizieren, wenn es einen Pfad im CSD vom Sender zum Empfänger gibt. Die Möglichkeiten für indirekte Kommunikation können daher z. B. durch eine Tiefensuche im CSD von den möglichen Sendern aus (oder in Gegenrichtung der Kanten von den Empfängern aus) ermittelt werden.

Die in Abschnitt 3.3.4 beschriebene Erzeugung neuer Kandidaten muss so angepasst werden, dass der aktuell modifizierte Controller C nur dann Synchronisationsnachrichten von einem besser informierten Controller C' erhält, wenn Nachrichten direkt oder indirekt von C' an C gesendet werden können. Um die infrage kommenden Controller zu ermitteln, wird eine Tiefensuche im CSD vom Empfänger der Synchronisation C aus entgegengesetzt der Richtung der Konnektoren durchgeführt. Die Tiefensuche berücksichtigt dabei nur Systemobjekte. Durch die Tiefensuche wird auch ein Pfad von C' zu C bestimmt, über den die Synchronisationsnachrichten gesendet werden können.

Einer der so ermittelten Controller wird als C' ausgewählt. Dann werden entsprechende Transitionen zum Durchführen der Synchronisation auf dem Pfad von C' zu C angelegt. Der Algorithmus erzeugt dann einen Kandidaten, der eine Kette von Synchro-

nisationen enthält (vgl. Abbildung 3.18). In jedem der Controller eines Objekts auf dem Pfad wird eine Transition angelegt, die eine Synchronisationsnachricht vom Vorgänger empfängt. Vom Folgezustand dieser Transition aus wird eine weitere Transition angelegt, die eine neue Synchronisationsnachricht zum Nachfolger sendet. Für Sender und Empfänger selbst wird nur jeweils eine Transition angelegt, da ersterer keinen Vorgänger und letzterer keinen Nachfolger auf dem Pfad hat.

Ist kein möglicher Sender erreichbar, so kann C nicht genügend informiert werden, um den ausgewählten SG-Zustand s durch Senden einer Nachricht implementieren zu können (siehe Abschnitt 3.3.4). Dadurch muss der Algorithmus eine andere Alternative suchen (Backtracking). Gibt es keine solche, so ist die MSD-Spezifikation aufgrund der architekturbedingten Einschränkungen der Kommunikation nicht realisierbar.

Die Zustandskorrespondenzen werden wie beim Hinzufügen einzelner Synchronisationen nach der Definition für `corr` in Abschnitt 3.3.3 berechnet. Entsprechend gilt nach Austausch aller Synchronisationsnachrichten der Kette, dass der Controller jedes der Objekte auf dem Pfad in einem Zustand ist, in dem seine korrespondierenden Zustände eine Teilmenge der korrespondierenden Zustände des aktuellen Zustands im Controller des Senders sind. Dies gilt auch für den Empfänger, der damit (mindestens) genauso gut informiert ist, als hätte er die Synchronisation direkt vom Sender empfangen.

Es ist möglich, dass die Controller derjenigen Objekte, die die Synchronisationsnachrichten weiterleiten, in ihren Zuständen vor Empfang der Synchronisationsnachricht bereits zur Unterscheidung von SG-Zuständen in der Lage waren, die der ursprüngliche Sender nicht unterscheiden konnte. In diesem Fall ist diese Unterscheidung durch die Schnittmengenbildung bei der Berechnung der Korrespondenzmengen auch allen nachfolgenden Objekten auf dem Pfad möglich, einschließlich des (endgültigen) Empfängers. Dadurch können durch eine solche Kette von Synchronisationen im Einzelfall weitere Synchronisationen, die sonst später erforderlich würden, vermieden werden. Der Algorithmus berücksichtigt diesen Fall zwar, führt ihn jedoch nicht gezielt herbei.

Anpassung der Implementierung von Nachrichten der Spezifikation

Die beschriebene Erweiterung stellt sicher, dass zusätzliche Synchronisationsnachrichten die definierten Kommunikationseinschränkungen einhalten. Zusätzlich muss jedoch sichergestellt werden, dass in der Spezifikation vorgesehene Nachrichten ebenfalls nur unter den definierten Einschränkungen gesendet werden. Ereignisse, die diesen widersprechen müssen bei der Erzeugung des SG (bzw. bei der Interpretation der MSD-Spezifikation) speziell behandelt werden: Handelt es sich um ein Systemereignis, so darf es nicht implementiert werden. Handelt es sich um ein Umgebungsereignis, so kann es ignoriert werden. In beiden Fällen wird bei der Erzeugung des SG im entsprechenden SG-Zustand keine ausgehende Transition für das Ereignis angelegt. Dies wiederum kann Auswirkungen darauf haben, ob ein gegebener SG-Zustand implementiert ist.

Einschränkungen der Kommunikation durch die Architektur können somit dazu führen, dass eine sonst nicht implementierbare MSD-Spezifikation implementierbar wird, weil bestimmte problematische Umgebungsnachrichten nicht auftreten können. Umgekehrt können jedoch auch die einzigen implementierbaren kontrollierbaren Ereignisse durch architekturbedingte Einschränkungen der Kommunikation verboten werden.

3.5 Zusammenfassung

In diesem Kapitel wurde ein Synthesealgorithmus vorgestellt, welcher für eine gegebene MSD-Spezifikation ein verteiltes Implementierungsmodell erzeugt, das aus je einem Controller (siehe Abschnitt 2.2) als Verhaltensmodell für jedes der Systemobjekte besteht. Sofern die Spezifikation implementierbar ist, konstruiert der Algorithmus so lange Kandidaten für diese Controller, bis sie zusammengenommen die MSD-Spezifikation erfüllen. Ist keine Implementierung möglich, so meldet der Algorithmus dies dem Benutzer.

Zunächst wurde erläutert, wie der Zustandsraum einer MSD-Spezifikation als Graph repräsentiert werden kann und was in diesem Kontext eine Lösung des Problems der verteilten Synthese ist. Anschließend wurde der Algorithmus gegen Vorarbeiten an demselben Lehrstuhl und in Kooperation entstandene Arbeiten abgegrenzt. Als Hauptbeitrag des Kapitels wurde die Basisvariante des Algorithmus vorgestellt und an einem Beispiel erläutert. Schließlich wurde eine Erweiterung des Algorithmus diskutiert, die die Berücksichtigung von architekturbedingten Einschränkungen der Kommunikation erlaubt. Diese stellt sicher, dass zusätzliche Kommunikation zwischen Subsystemen nur dann erzeugt wird, wenn sie in der definierten Architektur tatsächlich möglich ist.

Die wichtigste Besonderheit des hier vorgestellten Algorithmus gegenüber anderen Ansätzen (siehe Kapitel 7) ist, dass dieser auch dann eine Implementierung der gegebenen MSD-Spezifikation erzeugt, wenn dazu zusätzliche Kommunikation der Objekte erforderlich ist, die in den MSDs nicht explizit definiert wird. In solchen Fällen erzeugt der Algorithmus das zusätzliche Kommunikationsverhalten automatisch. Die Identifikation der zusätzlich erforderlichen Nachrichten wird dadurch möglich, dass das lokale Wissen der Objekte darüber berücksichtigt wird, welchem globalen Zustand der Spezifikation der aktuelle lokale Zustand des Objekts entspricht. Der Algorithmus erzeugt dieses zusätzliche Verhalten nur dann, wenn eines der Objekte ohne zusätzliche Informationen von anderen Objekten nicht gemäß der Spezifikation agieren kann.

Die Implementierung der Konzepte dieses Kapitels wird in Abschnitt 6.1 vorgestellt. Eine grundlegende Evaluierung durch Ausführung dieser Implementierung auf Beispielen wird in Abschnitt 6.2 diskutiert.

Wesentliche Einschränkungen des hier beschriebenen Algorithmus sind die fehlende Berücksichtigung von Echtzeitanforderungen, der Übertragungszeiten von Nachrichten und asynchroner Kommunikation. Diese Einschränkungen werden jedoch im nachfolgenden Kapitel 4 durch eine entsprechend erweiterte Variante des Algorithmus aufgehoben.

Kapitel 4

Synthese verteilter Echtzeitsysteme

Dieses Kapitel stellt eine Erweiterung des verteilten Syntheseverfahrens aus Kapitel 3 vor, die dieses für Echtzeitsysteme anwendbar macht. Dazu muss das Verfahren zeitbehaftete Systemanforderungen und Umgebungsannahmen unterstützen, wie sie mittels der in Abschnitt 2.1.5 beschriebenen zeitbehafteten Elemente in MSDs modelliert werden können. Durch diese Elemente können die zu einem gegebenen Zeitpunkt erlaubten Systemnachrichten und die möglichen Umgebungsnachrichten von den zeitlichen Abständen zwischen früheren Nachrichten abhängen. Somit ist auch ein Implementierungsmodell nur dann korrekt, wenn es diese Zeitabstände einhält. Die zeitbehaftete verteilte Synthese erzeugt daher als Ergebnis für jedes Systemobjekt einen zeitbehafteten Controller (siehe Abschnitt 2.2.2). Ein zeitbehafteter Controller kann für unterschiedliche Zeiträume zwischen empfangenen Nachrichten unterschiedliche Reaktionen auf diese vorsehen und auch selbst zeitgesteuert Nachrichten senden.

Für die Synthese auf Basis von zeitbehafteten MSDs sind zwei wesentliche konzeptionelle Erweiterungen erforderlich:

- Der ursprüngliche Algorithmus basiert auf einer Repräsentation der MSD-Spezifikation als Graph, dem Spezifikationsgraphen (SG) (siehe Abschnitt 3.2.1). Zur Unterstützung zeitbehafteter MSDs muss daher auch dieser Graph um Echtzeitverhalten erweitert werden, um zeitabhängig unterschiedliche Anforderungen und Annahmen repräsentieren zu können.

- Der Synthesealgorithmus selbst muss dann so erweitert werden, dass er auf Basis dieses erweiterten SG nur sicheres zeitbehaftetes Kommunikationsverhalten erzeugt. Dazu muss die Unterscheidung zwischen sicheren und unsicheren Zeitbereichen für Nachrichten explizit modelliert und bei der Synthese beachtet werden.

In einem weiteren Schritt wird das zeitbehaftete Syntheseverfahren nochmals erweitert, um asynchrone Kommunikation zu unterstützen und um die Übermittlungsdauer von Nachrichten berücksichtigen zu können.

Das Kapitel ist wie folgt unterteilt: Zunächst stellt Abschnitt 4.1 eine Echtzeitvariante des Anwendungsbeispiels der Produktionszelle aus Abschnitt 1.1 vor. Abschnitt 4.2 stellt die zeitbehaftete Erweiterung des SG vor, die zur Umsetzung der zeitbehafteten verteilten Synthese erforderlich ist. Abschnitt 4.3 beschreibt die Erweiterungen des eigentlichen Synthesealgorithmus und erläutert diese anhand einer beispielhaften Ausführung für das Anwendungsbeispiel. Der Echtzeit-Synthesealgorithmus wird in Abschnitt 4.4 um asynchrone Kommunikation mit Zeitbedarf der Nachrichtenübermittlung erweitert. Schließlich fasst Abschnitt 4.5 das Kapitel zusammen.

4.1 Erweitertes Anwendungsbeispiel

Hier wird die in Abschnitt 3.1 eingeführte MSD-Spezifikation der Produktionszelle um Echtzeitverhalten erweitert. Die Struktur des Systems bleibt dabei im Wesentlichen unverändert. Es werden nun aber konkrete Anforderungen an das Zeitverhalten des Systems in die Spezifikation aufgenommen. Dadurch kann auch relevant sein, welchen Zeitbedarf Aktionen von Umgebungsobjekten, wie beispielsweise das Pressen, haben.

Im Beispiel wird nun nicht mehr angenommen, dass der Pressvorgang nach Senden des Kommandos `press` bereits sofort abgeschlossen ist. Daher wird die Klasse PressController in der erweiterten Spezifikation um eine Operation `done` erweitert. Durch Senden der entsprechenden Nachricht meldet die Presse `p` ihrem Steuergerät die erfolgte Ausführung eines vorangegangenen Befehls `press`. Zusätzlich zu dieser Erweiterung wird die Spezifikation durch die Einführung neuer MSDs ergänzt, die zusätzliche Echtzeitanforderungen und -annahmen modellieren. Diese werden nachfolgend vorgestellt.

4.1.1 Anforderungen

Abbildung 4.1 zeigt die Requirement MSDs für die neuen Anforderungen **R6** und **R7**, die zu den bisherigen Anforderungen **R1** bis **R5** (siehe Abschnitt 3.1.2) hinzukommen. Im natürlichsprachlichen Text neben bzw. unter den MSDs wird die jeweilige Anforderung zusätzlich informell beschrieben.

Die Anforderung **R6** legt fest, dass das Steuergerät aC nach Senden des Befehls `blankToPress` an seinen Roboterarm `a` mindestens 3 Sekunden warten muss. Diese Zeit steht dem Arm zur Verfügung, um die geforderte Bewegung eines Rohlings zur Presse durchzuführen. Im MSD wird diese minimale Wartezeit durch die heiße Zeitbedingung mit unterer Schranke für die Uhr cBtP (Uhrenvariablen beginnen hier immer mit einem *c* für engl. *clock*, *BtP* steht für `blankToPress`) definiert. Erst wenn die Bedingung erfüllt ist, darf das Steuergerät des Arms durch die Nachricht `blankAtPress` dem Steuergerät pC der Presse die Ankunft des Rohlings mitteilen. Es wird bereits durch die (alte) Anforderung **R3** gezwungen, dies irgendwann zu tun.

Die Anforderung **R7** verlangt vom System, dass spätestens 30 Sekunden nach dem Erkennen eines neuen intakten Rohlings auf dem Zuführförderband dieser Rohling gepresst sein muss. Den Abschluss des Pressens meldet die Presse `p` mittels der Nachricht `done` an das Steuergerät der Presse. Das MSD modelliert diese Anforderung durch eine heiße Zeitbedingung mit einer oberen Schranke für die Uhr cIB (*IB* steht für `intactBlank`). Diese Uhr modelliert die Zeit, die seit der Erzeugung des aktiven MSDs und somit seit der Nachricht `intactBlank` vergangen ist. Die Nachricht `blankAtPress` dient in diesem MSD nur dazu, eine Reihenfolge der beiden anderen Nachrichten festzulegen; sie könnte auch durch eine heiße Bedingung mit dem Ausdruck `true` ersetzt werden.

Eine Besonderheit der Anforderung **R7** ist, dass sie nur durch rechtzeitiges Senden einer Umgebungsnachricht erfüllt werden kann. Sie kann daher nur dann erfüllt werden, wenn angenommen werden kann, dass die Umgebung diese Nachricht bis zu einem bestimmten Zeitpunkt senden wird. Das nachfolgend vorgestellte Assumption MSD formalisiert eine passende Umgebungsannahme, durch die die Spezifikation implementierbar wird. Zusammen mit dieser Annahme drückt die Anforderung **R6** aus, dass das

System seine Nachrichten jeweils früh genug senden muss, um auch dann ein Pressen innerhalb von 30 Sekunden zu garantieren, wenn die Umgebung `done` so spät sendet, wie die Annahmen gestatten.

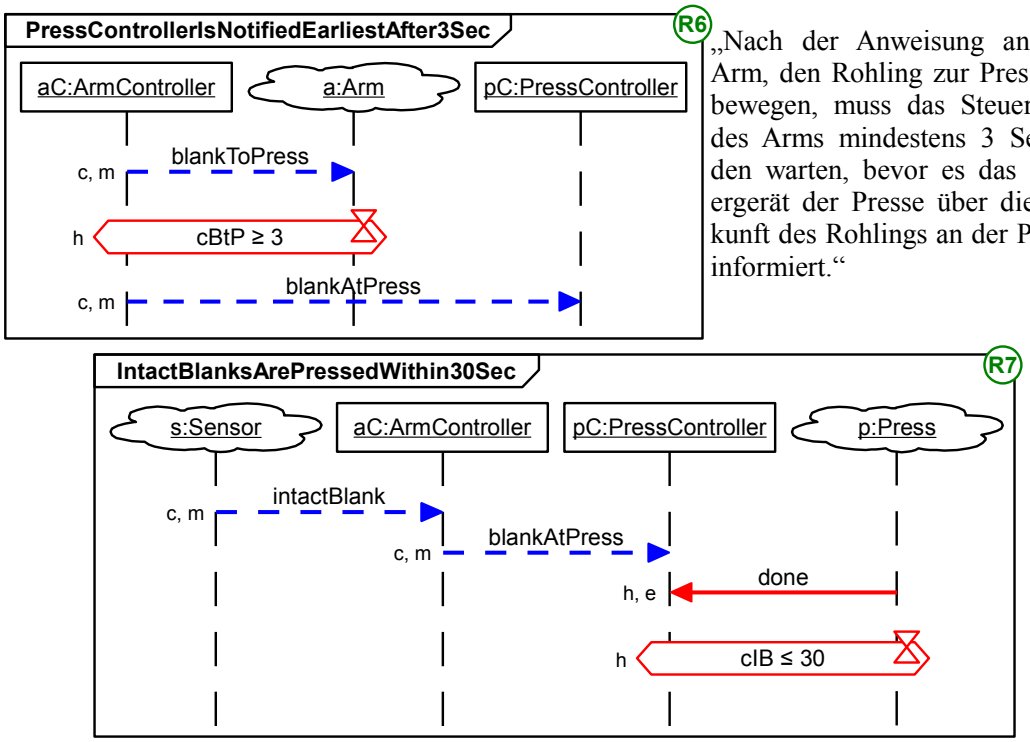


Abbildung 4.1: Die neuen Anforderungen für die zeitbehaftete Produktionszelle

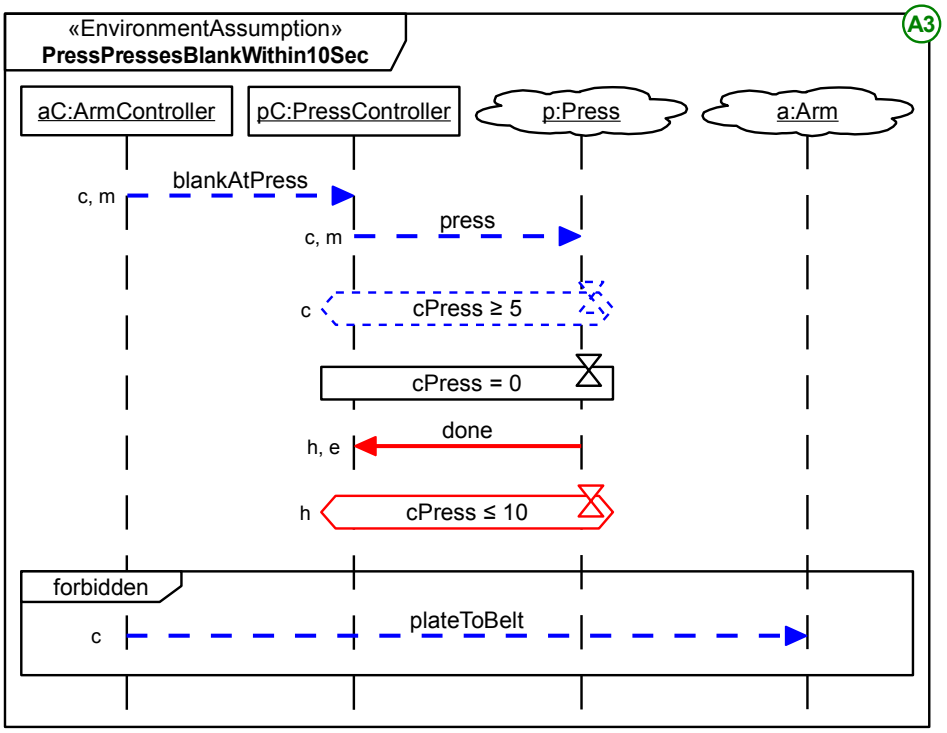
4.1.2 Umgebungsannahmen

Abbildung 4.2 zeigt das Assumption MSD für die neue Annahme **A3**, die zu den bisherigen Annahmen **A1** und **A2** (siehe Abschnitt 3.1.3) hinzukommt. Darunter ist die natürlichsprachliche Entsprechung der Annahme angegeben.

Die Annahme **A3** definiert, dass die Presse maximal 10 Sekunden zum Pressen benötigt und somit die Antwort `done` spätestens 10 Sekunden nach dem Befehl `press` folgen muss. Dies wird einerseits durch eine heiße Zeitbedingung modelliert, andererseits durch einen Clock Reset, der unmittelbar nach `press` durchgeführt wird (die kalte Zeitbedingung davor wird instantan ausgewertet). Vor dem Clock Reset modelliert die Uhr cPress stattdessen die Zeit seit `blankAtPress`.

Die Beschränkung von 10 Sekunden für das Pressen gilt jedoch nur dann, wenn die Presse zuvor seit dem Eintreffen des Rohlings genug Zeit hatte: Das MSD definiert, dass das System nach dem Senden der Nachricht `blankAtPress` mindestens 5 Sekunden

vor dem Befehl `press` vergehen lassen muss, damit die Zeitdauer für `done` überhaupt beschränkt wird. Dies wird durch die kalte Zeitbedingung nach `press` modelliert. Diese wird unmittelbar nach `press` ausgewertet und beendet das MSD durch eine kalte Violation, wenn sie früher als 5 Sekunden nach `blankAtPress` erreicht wird. Die kalte verbotene Nachricht `plateToBelt` führt zur sofortigen Terminierung des MSDs. So wird modelliert, dass die Annahme bei verfrühtem Senden der Nachricht nicht gilt.



„Wenn die Presse frühestens 5 Sekunden nach Ankunft eines Rohlings zum Pressen angewiesen wird, dann kann angenommen werden, dass sie innerhalb von 10 Sekunden nach dieser Anweisung den Abschluss des Pressvorgangs an ihr Steuergerät meldet.

Dies gilt jedoch nicht, wenn der Arm bereits vor Abschluss des Pressens angewiesen wird, die gepresste Metallplatte zum Ablageförderband zu bewegen.“

Abbildung 4.2: Die neue Annahme für die zeitbehaftete Produktionszelle

4.1.3 Hauptunterschiede zur Basisvariante der MSD-Spezifikation

Die beschriebenen zusätzlichen Anforderungen und Annahmen führen Zeitbedingungen in die Spezifikation ein, welche vom System und seiner Umgebung verlangen, bestimmte Nachrichten nicht vor bzw. nicht nach einem bestimmten Zeitpunkt zu senden. Im konkreten Fall wird einerseits die Einhaltung von Maximal- und Minimalzeiten für die (Re-)Aktionen des Systems gefordert. Andererseits wird angenommen, dass die Bearbeitung bestimmter Befehle durch die unkontrollierbaren Subsysteme Roboterarm und Presse jeweils nur einen bestimmten Zeitbedarf hat. Dabei wird der Zeitbedarf für das

Bewegen des Arms nur implizit angenommen, während der für das Pressen in einer eigenen Annahme modelliert wird. Um feststellen zu können, zu welchem Zeitpunkt die Presse den Befehl an sie bearbeitet hat, wurde die neue Nachricht `done` eingeführt. Diese Nachricht stellt hier das Bindeglied zwischen den Anforderungen und den Umgebungsannahmen dar.

In der Basisvariante der Spezifikation hatten die Controller der Systemobjekte prinzipiell beliebig viel Zeit, ihre Nachrichten zu senden. Durch die hier beschriebene Erweiterung der Spezifikation müssen sie nun jeweils so senden, dass eine Einhaltung der globalen Zeitbedingungen garantiert ist. Damit die Controller zeitabhängiges Verhalten beschreiben können, müssen diese als zeitbehaftete Controller, also in Form einer zeitbehafteten Variante von Automaten erzeugt werden. Sie müssen im konkreten Fall alle Befehle nach `intactBlank` so senden, dass `done` auch dann innerhalb von 30 Sekunden nach `intactBlank` auftritt, wenn die Umgebung die maximal angenommene Zeitdauer für diese Bestätigung benötigt.

Die Übertragungszeit der Nachrichten selbst wird in dieser Variante des Beispiels nicht modelliert. Dies geschieht erst in einer erweiterten Variante, die in Abschnitt 4.4.2 vorgestellt wird. Hier wird also weiterhin für alle Nachrichten angenommen, dass ihre Übertragungsdauer vernachlässigbar ist.

4.2 Erweiterung des Spezifikationsgraphen um Echtzeitverhalten

Dieser Abschnitt beschreibt, wie der in Abschnitt 3.2.1 eingeführte Spezifikationsgraph (SG) erweitert werden kann, um das mittels Timed MSDs (siehe Abschnitt 2.1.5) definierbare Echtzeitverhalten zu berücksichtigen. Die beschriebenen Konzepte erlauben direkt eine Play-out-Simulation [BGH+14] des definierten zeitbehafteten Verhaltens. Zur Synthese auf Basis zeitbehafteter Spezifikationen sind jedoch zusätzlich die in Abschnitt 4.3 erläuterten Erweiterungen des Algorithmus selbst erforderlich.

Der Inhalt dieses Abschnitts wurde teilweise bereits veröffentlicht [BGH+14], jedoch deutlich weniger ausführlich und auf ein anderes Beispiel bezogen. Zudem werden hier weitere Konzepte vorgestellt, die speziell für die Synthese relevant sind. Die hier beschriebenen Kernaspekte des zeitbehafteten SG wurden vollständig im Rahmen dieser Arbeit entwickelt. Die Evaluierung der Konzepte anhand von Beispielen erfolgte in Kooperation mit den Co-Autoren der erwähnten Veröffentlichung. Auf dieser Basis wurden mittlerweile weitere Konzepte zu zeitbehafteten MSDs entwickelt [HS14, KHD14].

In Abschnitt 4.2.1 wird die zeitbehaftete Variante des SG als Repräsentation des Zustandsraums einer zeitbehafteten MSD-Spezifikation vorgestellt. Anhand des Beispiels aus Abschnitt 4.1 werden die zeitbehafteten Erweiterungen in Abschnitt 4.2.2 erläutert. Anschließend werden diese Erweiterungen in Abschnitt 4.2.3 systematisch beschrieben und es wird näher erläutert, wie der zeitbehaftete SG mittels des Konzepts der *Clock Zones* berechnet werden kann. Da der erweiterte SG neue Arten von Ereignissen definiert, müssen für diese die Kriterien für Beobachtbarkeit und Kontrollierbarkeit (siehe Abschnitt 2.2.1) neu festgelegt werden. Dies geschieht in Abschnitt 4.2.4. Zusätzlich wird dort das neue Konzept der *Verhinderbarkeit* von Umgebungsereignissen eingeführt.

4.2.1 Der zeitbehaftete Spezifikationsgraph

Im nicht zeitbehafteten Fall wird jeder Zustand der Spezifikation durch die Cuts aller aktiven MSDs, die Belegungen der Diagramm- und Objektvariablen und die Bindungen der symbolischen Lifelines charakterisiert. Bei Verwendung von zeitbehafteten MSDs kommt die aktuelle Belegung der verwendeten Uhren hinzu. Da die Uhren jeweils beliebige reelle Werte annehmen können, würde eine explizite Repräsentation jeder einzelnen möglichen Uhrenbelegung durch einen eigenen Zustand jedoch zu einem unendlich großen Zustandsraum führen. Daher wurde zur *symbolischen* Repräsentation der möglichen Uhrenbelegungen das Konzept der *Clock Zones* eingeführt [Dil90, ACD93]. Diese fassen Uhrenbelegungen, für die das Systemverhalten identisch bleibt, zusammen.

MSD-Spezifikationen können neben dem Systemverhalten jedoch auch das Umgebungsverhalten modellieren, was bei der Berechnung der Clock Zones berücksichtigt werden muss. Der zeitbehaftete SG muss (wie der nicht zeitbehaftete SG) zwischen diesem Umgebungsverhalten und dem Systemverhalten unterscheiden können, da nur letzteres durch die Software des Systems kontrollierbar ist. Diese Unterscheidung zwischen kontrollierbarem und unkontrollierbarem Verhalten wird auch in *Timed Game Automata* [MPS95] gemacht, einer erweiterten Variante der Timed Automata. Wie die Semantik der Timed Automata wird auch die der Timed Game Automata auf Grundlage von Clock Zones definiert. Auf MSDs wurde das Konzept der Clock Zones jedoch noch nicht angewendet. Dies erfordert eine Anpassung, da MSDs Zeitbedingungen anders modellieren als Timed (Game) Automata.

Es existieren Ansätze, die durch Abbildung von MSDs (bzw. LSCs) auf Timed Game Automata indirekt eine Verwendung von Clock Zones ermöglichen [Gre11, LLNP10], diese Ansätze haben jedoch deutliche Einschränkungen. Da diese Abbildungen nur maximal eine Instanz jedes MSDs ermöglichen, können sie die in Abschnitt 2.1.2 beschriebene Semantik (die „invariante Interpretation“ von MSDs [HM03]), bei der mehrere aktive MSDs je MSD gleichzeitig existieren können, nicht umsetzen (siehe Abschnitt 3.2.4). Weiterhin werden dynamische Objektsysteme und symbolische Lifelines (vgl. Abschnitt 2.1.4) nicht unterstützt. Diese Arbeit verwendet daher keine dieser Abbildungen und beschreibt stattdessen in den folgenden Abschnitten, wie ein zeitbehafteter SG für MSDs bei direkter Verwendung von Clock Zones umgesetzt werden kann.

Gegenüber der nicht zeitbehafteten Variante ändert sich die Berechnung der Folgezustände für eine gegebene Kombination aus Ausgangszustand und Ereignis. Insbesondere kann die Auswirkung der Ereignisse in einem Zustand im zeitbehafteten Fall davon abhängen, zu welchem Zeitpunkt dieser Zustand betreten wird. Jedem SG-Zustand wird daher eine Clock Zone zugeordnet, welche die Uhrenwerte modelliert, die bei Erreichen des Zustands möglich sind. Diese Clock Zone wird hier als *initiale Zone* bezeichnet. Sie definiert genau die Uhrenwerte, mit denen der Zustand auf Pfaden vom Startzustand aus betreten werden kann.

Clock Zones haben bereits weite Verbreitung im Kontext der Analyse von Timed Automata erlangt [BY04]. Anders als bei Timed Automata ist aber bei Timed MSDs die Menge der Uhren nicht fest, sondern sie hängt von den aktuell aktiven MSDs ab, die jeweils lokale Uhrenvariablen definieren können. Die Menge von Uhren (bzw. Uhreninstanzen) kann sich daher von Zustand zu Zustand ändern.

Vor dem Senden von Nachrichten kann im Allgemeinen Zeit vergehen, was die initiale Zone des Folgezustands der betreffenden SG-Transition entsprechend erweitern kann. Durch Erreichen von Clock Resets in den aktiven MSDs können Uhren auf den Wert 0 zurückgesetzt werden, was die initiale Zone des Folgezustands gegenüber der des aktuellen Zustands entsprechend einschränkt. Durch Erreichen von Zeitbedingungen können – abhängig von den möglichen Uhrenwerten in der aktuellen initialen Zone – heiße oder kalte Violations auftreten.

Durch die neue Berechnung von Folgezuständen und die neuen Transitionen für Zeitentscheidungsereignisse muss auch die Definition einer Gewinnstrategie für das System (siehe Abschnitt 3.2.2) angepasst werden. Dabei bleibt das Hauptkriterium, dass das System immer wieder einen der Zielzustände erreichen muss, erhalten. Jedoch ist die Annahme der nicht zeitbehafteten Synthese, dass das System immer vor der Umgebung reagieren kann, nicht mehr sinnvoll: Zeitbedingungen können das System zum Warten zwingen, während die Umgebung senden kann. Auch wenn das System freiwillig wartet, darf nicht angenommen werden, dass die Umgebung dies ebenfalls tut. Eine Präzedenz von Systemnachrichten wird daher nur noch dann angenommen, wenn diese vor der frühesten Umgebungsnachricht gesendet werden können. Abschnitt 4.3.1 stellt die entsprechend angepasste Definition der zeitbehafteten Gewinnstrategie vor.

4.2.2 Illustration des zeitbehafteten Spezifikationsgraphen am Beispiel

Abbildung 4.3 zeigt rechts einen Ausschnitt des zeitbehafteten SG für das Anwendungsbeispiel. Links zeigt sie die MSDs für die Anforderung **R6** und die Annahme **A3** mit den möglichen Cuts. Im MSD zu **A3** wurde aus Platzgründen die Lifeline für Arm a ausgelassen. Zusätzlich zu diesen MSDs können weitere aktiv sein.

Neben jedem Cut werden die Nummern der SG-Zustände aufgeführt, die im jeweiligen MSD in diesem Cut sind. Die eingeklammerten Cuts sind bei konkreten Uhrenbelegungen nur Zwischenergebnisse, da die folgenden Bedingungen sofort ausgewertet werden und der Cut entsprechend voranschreitet. Bei der Betrachtung von Intervallen möglicher Uhrenbelegungen im SG kann es jedoch sein, dass von den konkreten Uhrenwerten innerhalb des Intervalls abhängt, ob eine erreichte Bedingung erfüllt ist oder nicht. Es gibt dann keinen eindeutigen Folgezustand für das gesamte Intervall. Dann müssen für diese Bedingung beide Möglichkeiten (erfüllt und nicht erfüllt) explizit behandelt werden, wie nachfolgend erläutert wird. In jedem Zustand des zeitbehafteten SG ist nach der Zustandsnummer die initiale Zone des Zustands vermerkt. Die initiale Zone ist (wie jede Clock Zone) eine Konjunktion von Vergleichen über Uhrenwerte bzw. Uhrendifferenzen (also den möglichen Zeitabständen der Uhren zueinander). Sie modelliert, dass unmittelbar nach Betreten des Zustands die Uhren alle Werte haben können, für die dieser Ausdruck wahr ist. Zusätzlich gilt natürlich die Einschränkung, dass Uhrenwerte nicht negativ sein können. In Abschnitt 2.2.2 werden Clock Zones genauer behandelt.

Durch die erste Nachricht `intactBlank` in Zustand S.0 wird das hier nicht gezeigte MSD zu Anforderung **R7** (Abbildung 4.1) aktiv, in dem die Uhr cIB definiert ist. Jede Instanz einer Uhr hat initial den Wert 0, womit die initiale Zone von Zustand S.1 durch `cIB = 0` charakterisiert ist. Durch `blankToPress` wird das MSD zu **R6** aktiv, wodurch Zustand S.2 erreicht wird. Hierdurch wird die neue Uhr cBtP erzeugt.

Im neu erzeugten aktiven MSD zu **R6** ist der Cut initial vor der heißen Zeitbedingung `cBtP` $\geq$ `3`. Allgemein gilt: Eine durch einen Cut erreichte Zeitbedingung, die durch alle aktuell möglichen Uhrenwerte erfüllt ist, wird wie eine gewöhnliche Bedingung *sofort* ausgewertet, also als Teil derselben SG-Transition. Könnte der Zustand S.2 hier also nur mit einem Wert `cBtP` $\geq$ `3` betreten werden, so würde der Cut über die Bedingung hinweg voranschreiten. Allgemein bezeichnen wir eine Bedingung, die in der initialen Zone eines Zustands für alle Uhrenwerte erfüllt oder für alle Uhrenwerte unerfüllt ist, als *entschiedene* Bedingung. Ist beides möglich, so handelt es sich um eine *unentschiedene* Bedingung. In diesem Fall ist die Zeitbedingung `cBtP` $\geq$ `3` in S.2 entschieden, da sie für `cBtP = 0` nicht erfüllt sein kann.

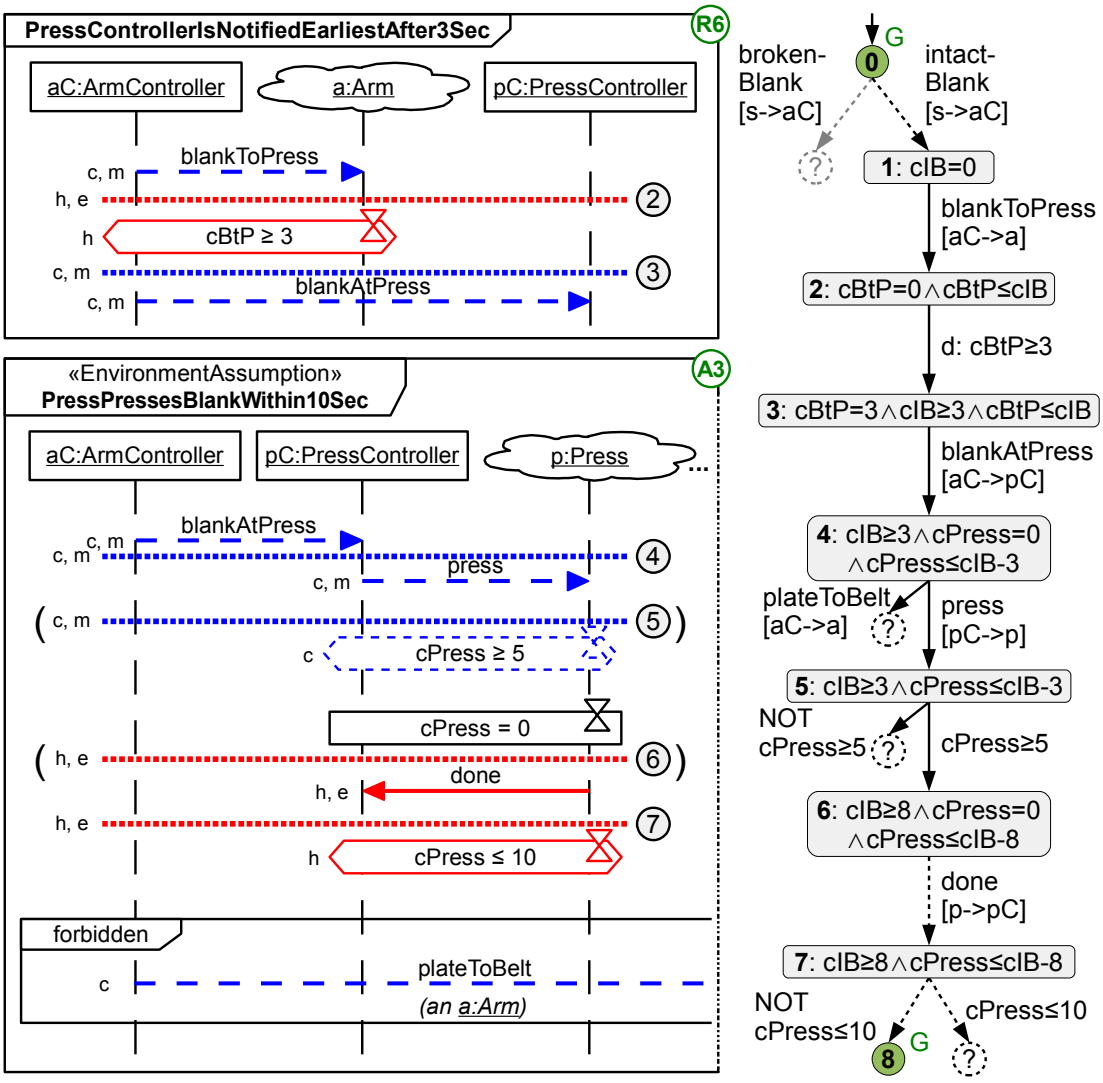


Abbildung 4.3: Ausschnitt aus dem zeitbehafteten SG (rechts) für das Beispiel mit den jeweiligen Cuts in zwei der Timed MSDs (links)

Wie in Abschnitt 2.1.5 erläutert wurde, führen unerfüllte heiße Zeitbedingungen mit unterer Schranke ($>$ oder $\geq$) nicht zu einer Violation. Das MSD verbleibt dann in einem heißen Cut vor der Bedingung, bis diese durch das Vergehen von genügend Zeit erfüllt ist (oder bis eine Violation des MSDs durch ein anderes Ereignis auftritt). Da sich der Cut durch Erfüllen der Bedingung ändert, muss dies zum Erreichen eines neuen Zustands im SG führen. Um diesen Spezialfall einer erfüllten Zeitbedingung durch Zeitablauf zu modellieren, können zeitbehaftete SG als neuen Ereignistyp *Verzögerungsereignisse* enthalten. Diese Ereignisse sind immer einer heißen Zeitbedingung mit unterer Schranke zugeordnet. Ihr Auftreten modelliert den Fall, dass genug Zeit vergangen ist, um die Bedingung zu erfüllen. Eine mit einem Verzögerungsereignis annotierte Transition führt daher in einen Folgezustand, in dem der Cut über die erfüllte Bedingung hinaus weitergeschaltet wurde.

Im Beispiel in Abbildung 4.3 ist daher für Zustand S.2 eine ausgehende Transition für ein Verzögerungsereignis eingezeichnet. Diese Transition ist mit `d: cBtP` $\geq$ `3` beschriftet, wobei „d“ für „delay“ steht. Das Verzögerungsereignis tritt durch das Verstreichen von (genau) genügend Zeit auf und Zustand S.3 wird erreicht.

Das Senden von `blankAtPress` in Zustand S.3 führt einerseits zur Beendigung des MSDs zu **R6**, andererseits zur Erzeugung eines neuen aktiven MSDs für **A3**. Dadurch gibt es im erreichten Zustand S.4 die Uhr cBtP aus **R6** nicht mehr, während die neue Uhr cPress für **A3** erzeugt wird.

Durch das Senden der Nachricht `press` in Zustand S.4 wird in Zustand S.5 eine kalte Zeitbedingung `cPress` $\geq$ `5` im MSD zu **A3** erreicht. Da cPress in Zustand S.5 nur durch den nach oben unbegrenzten Wert von cIB beschränkt wird, ist die Zeitbedingung in der initialen Zone dieses Zustands unentschieden. Sie ist für `0` $\leq$ `cPress` $<$ `5` verletzt, für `cPress` $\geq$ `5` jedoch erfüllt. Für die Uhrenwerte, mit denen Zustand S.5 erreicht werden kann, sind also beide Fälle möglich, führen jedoch jeweils zu unterschiedlichen Folgezuständen: Bei verletzter Zeitbedingung terminiert das MSD aufgrund einer kalten Violation, während bei erfüllter Bedingung der Cut voranschreitet, die Uhr cPress zurückgesetzt wird und die Nachricht `done` erreicht wird.

Existieren in SG-Zuständen – wie in Zustand S.5 – Cuts vor unentschiedenen Zeitbedingungen, die keine heißen Zeitbedingungen mit unterer Schranke sind, dann müssen beide möglichen Auswertungen dieser Bedingungen (erfüllt/unerfüllt) separat betrachtet werden. Um beide Fälle zu repräsentieren, können zeitbehaftete SG daher *Zeitentscheidungsereignisse* enthalten. Für jede unentschiedene Bedingung in einem Zustand wird ein Zeitentscheidungsereignis für den Fall der erfüllten Bedingung erzeugt und eines für den Fall der unerfüllten Bedingung. Eine Transition, die mit einem Zeitentscheidungsereignis für eine erfüllte Bedingung annotiert ist, führt in einen Folgezustand, in dem der Cut über diese Bedingung hinaus weitergeschaltet wurde. Eine Transition, die mit einem Zeitentscheidungsereignis für eine unerfüllte Bedingung annotiert ist, führt in einen Folgezustand, der durch die Verletzung dieser Bedingung entsteht.

Es gibt daher zwei ausgehende Transitionen in Zustand S.5, die mit den Zeitentscheidungsereignissen `cPress` $\geq$ `5` für die erfüllte Bedingung und `NOT cPress` $\geq$ `5` für die nicht erfüllte beschriftet sind. Da Zeitbedingungen sofort und vor allen anderen Ereignissen ausgewertet werden (siehe Abschnitt 2.1.5), sind andere Ereignisse in S.5 noch nicht relevant.

Um den Folgezustand einer Transition zu einem Zeitentscheidungsereignis zu berechnen, wird die initiale Zone des Ausgangszustands mit der Bedingung bzw. (für den Fall der unerfüllten Bedingung) mit deren Negation geschnitten. In den resultierenden Zeitbereichen ist die zuvor unentschiedene Bedingung zwangsläufig entschieden und wird sofort ausgewertet. Im dargestellten SG wird nur der Fall mit `cPress` $\geq$ `5` evaluiert. Unmittelbar nach der Zeitbedingung gilt dann als Zwischenergebnis der Berechnung des nächsten Zustands `cPress` $\geq$ `5`, durch den Clock Reset wird S.6 jedoch mit `cPress = 0` erreicht.

In Zustand S.6 kann als einziges Nachrichtenereignis `done` von der Umgebung ausgelöst werden. Die Umgebung kann zuvor beliebig lange warten; sie verletzt jedoch später in Zustand S.7 die Annahme **A3**, wenn durch zu langes Warten `cPress > 10` gilt.

Die Behandlung der heißen Zeitbedingung `cPress` $\leq$ `10` in Zustand S.7 erfolgt analog zur Behandlung von `cPress` $\geq$ `5` in S.5. Auch hier werden also Zeitentscheidungsereignisse angelegt. Im Beispiel führt ein Senden von `done` bei `cPress > 10` zu einer Verletzung der Annahme **A3**, da dann die heiße Zeitbedingung `cPress` $\leq$ `10` niemals erfüllt werden kann und somit irgendwann zwangsläufig eine Liveness Violation verursacht. Der dadurch erreichte Zustand S.8 ist daher ein Zielzustand. Er wird in einer Umgebung, die den Annahmen genügt, jedoch nie erreicht.

4.2.3 Repräsentation des zeitbehafteten Verhaltens von Timed MSDs mittels Clock Zones

Dieser Abschnitt beschreibt die Konzepte der zeitbehafteten Erweiterung des SG, insbesondere die konkrete Verwendung der Clock Zones. Im vorherigen Abschnitt wurde diese Erweiterung bereits am Beispiel vorgestellt; sie wird hier nun systematischer und detaillierter beschrieben. Dazu wird zuerst die Erzeugung des SG allgemein beschrieben, dann speziell die Behandlung von Zeitbedingungen. Schließlich wird die konkrete Berechnung der initialen Zones erläutert. Dabei werden die in Abschnitt 2.2.2 behandelten Operationen auf Clock Zones als bekannt vorausgesetzt.

Erzeugung des zeitbehafteten Spezifikationsgraphen

Der zeitbehaftete SG wird, wie der nicht zeitbehaftete, vom Startzustand aus durch Exploration derjenigen Ereignisse in den bereits besuchten Zuständen erzeugt, die eine Änderung des SG-Zustands bewirken können. Neben Nachrichtenereignissen kommen im zeitbehafteten Fall *Zeitereignisse* hinzu, die zeitabhängig ausgelöst werden. Die initiale Zone des Startzustands des SG ist immer die unbeschränkte Clock Zone `true`. Der Grund dafür ist, dass im Startzustand zunächst keine aktiven MSDs existieren und damit auch keine Uhrenvariablen. Im Rahmen der Exploration wird die initiale Zone jedes neu erzeugten Zustands aus der Zone des Vorgängerzustands berechnet. Das Ereignis, das zum Erreichen des neuen Zustands führt, kann dabei diese Berechnung beeinflussen – etwa wenn es die Ausführung eines Clock Resets bewirkt, oder wenn es sich um ein Zeitereignis handelt.

Bei der Erzeugung des SG wird die initiale Zone eines SG-Zustands als Bestandteil des Zustands aufgefasst. Dadurch ergibt sich, dass zwei SG-Zustände nur dann als identisch

betrachtet werden, wenn auch ihre initialen Zones identisch sind. Es können also mehrere Zustände existieren, bei denen die nicht zeitbehafteten Bestandteile (also die Bindungen der Lifelines, die aktuellen Cuts aller aktiven MSDs und die Belegungen aller Variablen, vgl. Abschnitt 3.2.1) identisch sind und nur die initiale Zone unterschiedlich ist.

Zur Berechnung einer neuen initialen Zone für einen neuen Zustand wird zunächst die existierende Zone des Vorgängerzustands kopiert und dann entsprechend des zum Zustandswechsel führenden Ereignisses modifiziert. Nicht nur die neuen Zeitereignisse können diese Kopie verändern, sondern auch Nachrichtenereignisse. Da in Timed MSDs vor dem Senden einer Nachricht grundsätzlich beliebig viel Zeit vergehen darf, werden bei Nachrichtenereignissen zunächst alle oberen Schranken der initialen Zone im Folgezustand entfernt.

Durch die Bearbeitung einer Nachricht in einem aktiven MSD können jedoch neue aktive MSDs entstehen oder existierende durch eine Violation beendet werden. Dadurch kann sich die Menge der Uhren im Folgezustand ändern. Weiterhin können Clock Resets und entschiedene Zeitbedingungen erreicht werden, die dann ebenfalls vor Betreten des neuen SG-Zustands behandelt werden. Diese können die initiale Zone des Zustands also zusätzlich beeinflussen.

Wird in einem SG-Zustand ein neues aktives MSD erzeugt, so werden neue Instanzen aller in diesem MSD verwendeten Uhren erzeugt. Da die Uhren eines MSDs für jedes seiner aktiven MSDs eigene Werte haben können (siehe Abschnitt 2.1.5), müssen diese durch die Clock Zone auch als eigenständig behandelt werden. Die initiale Zone des Zustands muss also um diese Uhren erweitert werden. Da Uhrenvariablen bei ihrer Erzeugung immer den Wert 0 haben, werden die unteren und oberen Schranken für diese Uhren in der initialen Zone jeweils auf 0 gesetzt. Werden in einem SG-Zustand zuvor aktive MSDs beendet, so werden die Uhren dieser aktiven MSDs nicht in die initiale Zone des Zustands übernommen.

Werden bei der Bearbeitung eines Nachrichtenereignisses Clock Resets erreicht, so wird der Cut in den betreffenden aktiven MSDs entsprechend modifiziert. In der initialen Zone des Folgezustands werden die unteren und oberen Schranke für diese Uhren jeweils auf 0 gesetzt. Dasselbe gilt für die Uhrendifferenzen dieser zurückgesetzten Uhren zueinander. Dies modelliert, dass sie alle zu demselben Zeitpunkt zurückgesetzt wurden, nämlich dem Zeitpunkt des Nachrichtenereignisses.

Zeitentscheidungsereignisse und Verzögerungsereignisse

Die in Abschnitt 4.2.2 am Beispiel eingeführten neuen Ereignistypen werden hier systematischer und detaillierter erläutert.

Wenn bei der Bearbeitung eines Nachrichtenereignisses Zeitbedingungen erreicht werden, dann wird zunächst jeweils geprüft, ob diese für die aktuell möglichen Uhrenwerte erfüllt sind, oder nicht. Aktuell möglich sind diejenigen Uhrenwerte, welche gemäß der bisherigen Berechnung in der neuen initialen Zone liegen. Nun sind für jede Zeitbedingung zwei Fälle zu unterscheiden: Sie kann entweder für alle möglichen Uhrenwerte erfüllt oder verletzt sein, oder sie ist nur für manche Uhrenwerte erfüllt, für andere aber verletzt. Im ersten Fall bezeichnen wir die Zeitbedingung als *entschieden*, im zweiten Fall als *unentschieden*.

Entsprechend wird der neue SG-Zustand berechnet: Ist eine entschiedene Zeitbedingung erfüllt, so wird der Cut entsprechend weitergeschaltet. Dies geschieht *sofort*, es wird also kein Zwischenzustand im SG erzeugt, in dem sich der Cut vor der Zeitbedingung befindet. Die initiale Zone bleibt dann unverändert.

Ist eine entschiedene Zeitbedingung verletzt, dann wird sie bei der Berechnung des neuen SG-Zustands abhängig davon behandelt, um was für eine Art von Bedingung es sich handelt: Handelt es sich um eine kalte Zeitbedingung, so tritt sofort eine kalte Violation auf. Handelt es sich um eine heiße Zeitbedingung mit oberer Schranke (also mit $<$ oder $\leq$), dann kann die Bedingung nicht mehr erfüllt werden (da Uhren nur im Wert steigen). Die Anforderungen (bzw. Annahmen bei Assumption MSDs) können dann durch die zwangsläufige Liveness Violation als verletzt betrachtet werden – dieser Fall wird dann wie eine heiße Violation behandelt. Sind jedoch verbotene kalte Nachrichten (siehe Abschnitt 2.1.4) in demselben MSD definiert, dann kann deren Auftreten noch eine Liveness Violation vermeiden. In diesem Spezialfall bleibt der Cut im neuen SG-Zustand vor der entschiedenen Bedingung stehen.

Verzögerungsereignisse Handelt es sich bei der entschiedenen verletzten Zeitbedingung um eine heiße Zeitbedingung mit unterer Schranke ($>$ oder $\geq$), dann wird sie als spezielles Ereignis im neuen SG-Zustand repräsentiert, das als *Verzögerungsereignis* bezeichnet wird. Der Cut bleibt dann zunächst vor der Bedingung stehen, die in diesem Spezialfall immer durch Warten erfüllt werden kann.

Das Erfüllen der Bedingung durch Warten wird dann durch das Auftreten des Verzögerungsereignisses modelliert. Existieren in einem Zustand Verzögerungsereignisse, so kann in diesem Zustand nicht länger gewartet werden als bis die Bedingung dieses Ereignisses erfüllt ist. Dies gilt, weil eine erfüllte Zeitbedingung nach der MSD-Semantik ohne Zeitverlust behandelt werden muss. Der Folgezustand wird dann zu genau dem Zeitpunkt betreten, zu dem die Bedingung erfüllt wird.

Gibt es in einem SG-Zustand mehrere Verzögerungsereignisse, so kann es sein, dass einige davon immer erst nach bestimmten anderen auftreten können. Die später auftretenden Ereignisse implizieren dann die früheren Ereignisse und bewirken, dass auch deren Bedingungen erfüllt sind und der Cut entsprechend weitergeschaltet wird. Für die späteren Ereignisse müssen dann erst in später erreichten Zuständen Transitionen angelegt werden. Beispielsweise muss eine Bedingung `c > 1` zwangsläufig vor `c > 2` erfüllt sein, wodurch `c > 2` erst im Folgezustand einer Transition für `d: c > 1` relevant wird.

Ein Verzögerungsereignis ist dann *früher* als ein anderes, wenn seine Bedingung eine schwächere untere Schranke für dieselbe Uhr definiert. Es ist auch dann früher, wenn die Bedingung sich auf eine andere Uhr bezieht und die Uhrendifferenzen implizieren, dass die Bedingung früher erfüllt sein muss. Für eine Clock Zone oder Zeitbedingung x kann mittels der Funktion `earlier` berechnet werden, ob diese in einem SG-Zustand s mit initialer Zone `initial(s)` früher als eine Clock Zone oder Zeitbedingung y ist:

$$earlier(s, x, y) = (x^{\uparrow} - y^{\uparrow}) \wedge initial(s)^{\uparrow} \neq false. \tag{4.1}$$

Die Funktion gibt `true` zurück, wenn es im Zustand s einen Zeitraum ($initial(s)^{\uparrow}$) geben kann, in dem x erfüllt ist, aber noch nicht y ($x^{\uparrow} - y^{\uparrow}$). Sonst gibt sie `false` zurück.

Handelt es sich bei der verletzten entschiedenen Zeitbedingung aber um eine kalte Zeitbedingung oder eine heiße Zeitbedingung mit oberer Schranke, dann verursacht diese eine Verletzung des aktiven MSDs entsprechend ihrer Temperatur. Dann werden bei einer kalten Verletzung die betroffenen aktiven MSDs beendet und bei einer heißen Verletzung wird eine Verletzung der Anforderungen bzw. Annahmen festgestellt. Neben diesen Änderungen des neuen SG-Zustands selbst wird dessen initiale Zone nicht direkt durch die Zeitbedingung verändert. Indirekt kann es aber beispielsweise bei einer kalten Verletzung eines aktiven MSDs zu Änderungen kommen, da die neue initiale Zone dann nicht mehr die Uhren des beendeten aktiven MSDs enthält.

Eine unentschiedene Zeitbedingung wird immer als eigenständiges Ereignis im neuen SG-Zustand behandelt, d. h. der Cut bleibt zunächst vor der Bedingung stehen. Die initiale Zone ändert sich nicht. Für heiße Zeitbedingungen mit unterer Schranke werden auch hier Verzögerungsereignisse angelegt. Da die Zeitbedingung unentschieden ist, kann sie bei Erreichen bereits erfüllt sein. Der Folgezustand kann daher zu jedem Zeitpunkt der initialen Zone des Ausgangszustands betreten werden, in dem die Bedingung erfüllt ist – nicht nur zum frühest möglichen Zeitpunkt wie bei verletzter Zeitbedingung.

Zeitentscheidungsereignisse Ist die unentschiedene Zeitbedingung kalt oder eine heiße Zeitbedingung mit oberer Schranke, dann wird sie durch ein Paar von *Zeitentscheidungsereignissen* repräsentiert: Eines von diesen repräsentiert den Fall, dass die Bedingung erfüllt ist, das andere den Fall einer Verletzung. Zeitentscheidungsereignisse haben vor anderen Ereignissen (also Nachrichtenereignissen und Verzögerungsereignissen) Vorrang; es können also in einem Zustand mit Zeitentscheidungsereignissen nur diese auftreten. Dies modelliert die Besonderheit, dass diese Ereignisse ohne Verzögerung behandelt werden (vgl. Abschnitt 2.1.5), während vor den anderen Ereignissen jeweils beliebig viel Zeit vergehen kann. Zeitentscheidungsereignisse und Verzögerungsereignisse werden hier auch als *Zeitereignisse* zusammengefasst. Ein SG-Zustand mit Zeitentscheidungsereignissen wird auch als *Zeitentscheidungszustand* bezeichnet.

Ein Zeitentscheidungsereignis tritt in einem SG-Zustand auf, wenn es in der initialen Zone dieses Zustands Uhrenwerte gibt, die die Bedingung erfüllen, und solche, die sie verletzen. Repräsentiert das Zeitentscheidungsereignis den Fall, dass die Bedingung erfüllt ist, dann wird die aktuelle initiale Zone mit der Bedingung geschnitten. Hierdurch wird der Zeitraum ermittelt, für den die Bedingung im Zustand, in dem das Ereignis aufgetreten ist, erfüllt ist. Repräsentiert das Zeitentscheidungsereignis jedoch eine Verletzung der Bedingung, so findet das Schneiden entsprechend mit der negierten Bedingung statt, um den Zeitraum der Verletzung zu ermitteln. Wir bezeichnen die so ermittelte Clock Zone als *ausgewählte Zone*.

Für die ausgewählte Zone ist die Zeitbedingung also entweder erfüllt oder verletzt. Die zuvor unentschiedene Zeitbedingung wird durch das Auftreten des Zeitentscheidungsereignisses daher zu einer entschiedenen. Der durch das Zeitentscheidungsereignis erreichbare Folgezustand und dessen initiale Zone werden deshalb so berechnet, wie es oben für entschiedene Zeitbedingungen beschrieben ist. Dabei wird die neue initiale Zone nicht auf Grundlage der vollständigen initialen Zone des Ausgangszustands, sondern auf Grundlage der ausgewählten Zone ermittelt.

Ob eine Zeitbedingung erfüllt ist, hängt davon ab, wann das Ereignis aufgetreten ist, das zum Erreichen der Bedingung geführt hat. Die Auswahl eines Zeitentscheidungsereignisses entscheidet somit praktisch nachträglich, in welchem Zeitraum das letzte Ereignis vor diesem stattgefunden hat – nämlich in dem der ausgewählten Zone. Letztendlich wird damit immer über den Sendezeitpunkt einer Nachricht entschieden, da Zeitereignisse immer relativ zu anderen Ereignissen stattfinden und somit nur Nachrichtenereignisse für die Erfüllung von Zeitbedingungen relevant sind.

Berechnung der initialen Zones

Eine wichtige Erweiterung des SG für den zeitbehafteten Fall ist die Berechnung der initialen Zones, insbesondere unter Berücksichtigung der neu hinzukommenden Ereignistypen Verzögerungsereignis und Zeitentscheidungsereignis. Diese Berechnung wird nachfolgend formal definiert.

Sei `initial(x)` die initiale Zone eines SG-Zustands x. Dann gilt für den Startzustand S.0 des SG: `initial(S.0)=true`. Bewirkt ein Ereignis `e` in einem beliebigen SG-Zustand s den Übergang zu einem SG-Zustand s', so kann `initial(s')` aus `initial(s)` wie folgt berechnet werden: Sei r die Menge aller Uhren, deren Clock Resets bei der Behandlung von `e` durch den Cut erreicht wurden. Sei V die Disjunktion der Bedingungen aller frühesten Verzögerungsereignisse in s (ermittelt durch `earlier`). Ist das Ereignis `e`

- ein Nachrichtenereignis, so gilt:
  $initial(s') = [r \mapsto 0]((initial(s)^{\uparrow} - V)$.
- ein Zeitereignis für eine erfüllte Bedingung c oder ein Verzögerungsereignis für eine unentschiedene Bedingung c, so gilt:
  $initial(s') = [r \mapsto 0](initial(s) \wedge c)$.
- ein Zeitentscheidungsereignis für eine verletzte Bedingung c, so gilt:
  $initial(s') = [r \mapsto 0](initial(s) \wedge \neg c)$.
- ein frühestes Verzögerungsereignis für eine verletzte Bedingung c, so gilt:
  $initial(s') = [r \mapsto 0](initial(s)^{\uparrow} \wedge earliest(c))$.

Der erste Fall modelliert, dass vor einem Nachrichtenereignis beliebig viel Zeit vergehen kann – mit einer Ausnahme: Die Entscheidung für ein Nachrichtenereignis statt eines der Verzögerungsereignisse bedeutet, dass die Bedingungen für letztere (also V) zum Sendezeitpunkt noch nicht erfüllt sein können. Der zweite und dritte Fall modellieren, dass die Entscheidung für eines der Zeitentscheidungsereignisse (bzw. Verzögerungsereignisse für unentschiedene Bedingungen) in s auch die Auswahl des betreffenden Teils (c bzw. $\neg c$) der initialen Zone von s bedeutet. Der letzte Fall modelliert ein Verstreichen von genau genügend Zeit, um die Bedingung des Verzögerungsereignisses zu erfüllen. Dabei sei $earliest(c)$ eine Zeitbedingung, die den frühesten Zeitpunkt modelliert, zu dem c erfüllt ist.

Wie oben beschrieben wurde, kann sich die Uhrenmenge in $initial(s')$ durch Erzeugen oder Beenden von aktiven MSDs ändern. Im ersten Fall sind die neu erzeugten Uhren dann jeweils gleich 0, im zweiten Fall entfallen aus $initial(s')$ alle Uhrenvergleiche, die sich auf die entfernten Uhren beziehen.

4.2.4 Beobachtbarkeit, Kontrollierbarkeit und Verhinderbarkeit

Für Nachrichtenereignisse gelten auch für die zeitbehaftete Synthese weiterhin die Definitionen aus Abschnitt 2.2.1. Da es in zeitbehafteten SG aber zusätzlich Zeitereignisse gibt (vgl. Abschnitt 4.2.3), müssen die Definitionen für Beobachtbarkeit und Kontrollierbarkeit nun auch für diese Ereignisse entsprechend definiert werden. Zusätzlich wird hier für Nachrichtenereignisse die neue Eigenschaft der *Verhinderbarkeit* eingeführt.

Beobachtbarkeit

Ob Zeitbedingungen erfüllt sind (und deren Zeitereignisse auftreten), hängt von den aktuellen Werten der durch die Bedingung verglichenen Uhren ab (siehe Abschnitt 2.1.5). Daher muss ein Controller, um das Auftreten eines Zeitereignisses feststellen zu können, den Wert der jeweiligen Uhr kennen. Dieser hängt vom Zeitpunkt des letzten Clock Resets der Uhr ab, beziehungsweise vom Zeitpunkt der Erzeugung des aktiven MSDs, wenn noch kein Clock Reset stattfand. Um den Wert einer Uhr zu kennen, muss ein Controller daher dasjenige Ereignis beobachten können, das zum letzten Clock Reset der Uhr bzw. zur Erzeugung des aktiven MSDs geführt hat. Im zweiten Fall handelt es sich um ein Auftreten der minimalen Nachricht des MSDs.

Eine Transition mit einem Zeitereignis kann auf mehreren unterschiedlichen Pfaden im SG erreicht werden, wodurch es mehrere letzte Clock Resets vor dem Zeitereignis geben kann. Zudem kann es mehrere mögliche letzte Ereignisse vor einem Clock Reset geben, wenn ein Zustand, in dem eine Uhr zurückgesetzt wird, mehrere eingehende Transitionen hat. Die oben genannte Bedingung für die Beobachtbarkeit des Ereignisses muss dann für jeden dieser Pfade gelten, d. h. der Controller muss alle infrage kommenden Ereignisse beobachten können. Nach diesen Kriterien wird für jede Uhr die Menge der Objekte ermittelt, die diese beobachten können. Diese Menge wird für die zeitbehafteten SG-Zustände gespeichert, in deren initialer Zone diese Uhr existiert. Wenn auf mehreren Pfaden ansonsten identische SG-Zustände erreicht werden, dann gelten diese Zustände bei unterschiedlichen Mengen beobachtbarer Uhren als unterschiedlich.

Bei Verzögerungsereignissen reicht die Kenntnis des letzten Clock Resets der verglichenen Uhr aus, um den Zeitpunkt des Ereignisses zu kennen: Dieser lässt sich direkt an der Bedingung ablesen. Bei Zeitentscheidungsereignissen ist es zwar notwendig, aber nicht hinreichend, den letzten Clock Reset zu kennen: Ein solches Ereignis tritt immer direkt (und ohne Zeitverlust) nach einem anderen Ereignis auf. Der Wert einer Uhr zum Zeitpunkt eines Zeitentscheidungsereignisses ist also der Zeitabstand zwischen dem letzten Clock Reset dieser Uhr und dem Auftreten des letzten vorhergehenden Ereignisses. Zur Beobachtbarkeit eines Zeitentscheidungsereignisses muss ein Controller also *sowohl* das letzte Ereignis vor dem letzten Reset, *als auch* das letzte Ereignis vor dem Zeitentscheidungsereignis beobachten können.

Definition 8 (Beobachtbarkeit von Zeitereignissen)**.** Ein Clock Reset (bzw. Erzeugen) einer Uhr ist *beobachtbar für einen Controller*, wenn dieser alle Ereignisse, die unmittelbar vor dem Clock Reset (bzw. Erzeugen) auftreten können, beobachten kann. Eine Uhr ist *beobachtbar für einen Controller in einem SG-Zustand*, wenn der Controller jeden letzten Clock Reset der Uhr vor Erreichen des Zustands beobachten kann. Ein

Verzögerungsereignis ist *beobachtbar für einen Controller*, wenn dieser die in der Bedingung verglichene Uhr in dem SG-Zustand beobachten kann, in dem dieses Ereignis auftritt. Ein Zeitentscheidungsereignis ist *beobachtbar für einen Controller*, wenn dieser im jeweiligen Zeitentscheidungszustand die verglichene Uhr beobachten kann *und* er alle Ereignisse eingehender Transitionen dieses Zustands beobachten kann.

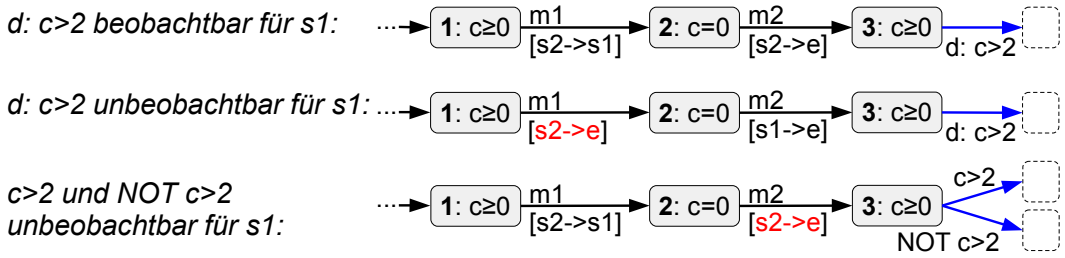


Abbildung 4.4: Beispiele für die Beobachtbarkeit von Zeitereignissen

Abbildung 4.4 zeigt je ein Beispiel für eine beobachtbares und für ein unbeobachtbares Verzögerungsereignis `d: c > 2`. Oben im Bild ist das Verzögerungsereignis beobachtbar für `s1`, da der letzte Clock Reset von `c` zusammen mit der durch `s1` empfangenen Nachricht `m1` stattfindet. Mittig im Bild kann der Zustand S.2 jedoch durch eine für `s1` unbeobachtbare Nachricht `m1` erreicht werden. Daher ist der letzte Clock Reset von `c` in Zustand S.3 für `s1` nicht bekannt und damit auch nicht der Zeitpunkt von `c > 2`. Somit ist das Ereignis `c > 2` unbeobachtbar. Unten im Bild wird ein ähnlicher Fall für eine kalte Zeitbedingung `c > 2` betrachtet. Hier sind die entsprechenden Zeitentscheidungsereignisse unbeobachtbar für `s1`, da dieses zwar den Clock Reset beobachten kann, aber nicht den Zeitpunkt des Betretens von S.3 kennt, weil `s1` `m2` nicht beobachten kann.

Die auf dem Begriff der Beobachtbarkeit basierende Definition für korrespondierende Zustände aus Abschnitt 3.3.3 kann ohne Änderungen für zeitbehaftete Systeme übernommen werden. Durch die Erweiterung dieses Begriffs auf Zeitereignisse berücksichtigt die Definition nun auch Transitionen mit diesen Ereignissen.

Kontrollierbarkeit

Auch die Kontrollierbarkeit von Zeitereignissen wird für Verzögerungsereignisse und Zeitentscheidungsereignisse unterschiedlich definiert: Verzögerungsereignisse können nur dann auftreten, wenn sowohl System als auch Umgebung zuvor keine Nachrichten senden. Dann jedoch treten sie zwangsläufig auf. Sie sind nur dann kontrollierbar, wenn im Ausgangszustand bis zur Erfüllung der Zeitbedingung ausschließlich kontrollierbare Ereignisse auftreten können. Das System kann dann durch Warten entscheiden, dass das früheste Verzögerungsereignis auftritt. Umgekehrt kann das System aber auch bei nicht kontrollierbaren Verzögerungsereignissen ihr Auftreten verhindern, sofern es im betreffenden Zustand mindestens ein kontrollierbares Nachrichtenereignis gibt.

Zeitentscheidungsereignisse treten abhängig vom Zeitpunkt des letzten Ereignisses vor ihnen *und* abhängig vom letzten Clock Reset der verglichenen Uhr auf. Solange das Sys-

tem seit dem letzten Clock Reset dieser Uhr eine Nachricht gesendet hat, hat es potentiell Einfluss darauf, welches der beiden Zeitentscheidungsereignisse auftritt. In diesem Fall werden diese Ereignisse dann als kontrollierbar betrachtet. Durch die Berechnung der unsicheren Zones kann sich später herausstellen, dass das System das alternative Zeitentscheidungsereignis nicht tatsächlich verhindern kann und dieses auch berücksichtigt werden muss. Die Klassifikation als „kontrollierbar“ ist hier also nur vorläufig.

Definition 9 (Kontrollierbarkeit von Zeitereignissen)**.** Ein Zeitereignis ist *kontrollierbar für einen Controller*, wenn es

- ein Verzögerungsereignis ist und vor ihm in demselben SG-Zustand mindestens ein kontrollierbares, aber keine unkontrollierbaren Ereignisse auftreten können, *oder*
- es sich um ein Zeitentscheidungsereignis handelt und der Controller seit dem letzten Clock Reset der von diesem referenzierten Uhr mindestens ein Nachrichtenereignis vor dem Zeitentscheidungsereignis kontrolliert.

Abbildung 4.5 zeigt einige Beispiele für die Kontrollierbarkeit von Zeitereignissen. Oben werden Verzögerungsereignisse behandelt: Im linken Beispiel gibt es nur eine kontrollierbare Transition in demselben Zustand wie das Verzögerungsereignis `d: c > 5`, wodurch $s1$ die Bedingung durch Warten erfüllen kann, alternativ aber auch vor dem Erfüllen der Bedingung senden kann. Im rechten Beispiel gibt es eine unkontrollierbare Transition, die beim Abwarten von $s1$ schalten kann. Daher kann $s1$ das Auftreten von `d: c > 5` nicht erzwingen und dieses ist daher unkontrollierbar.

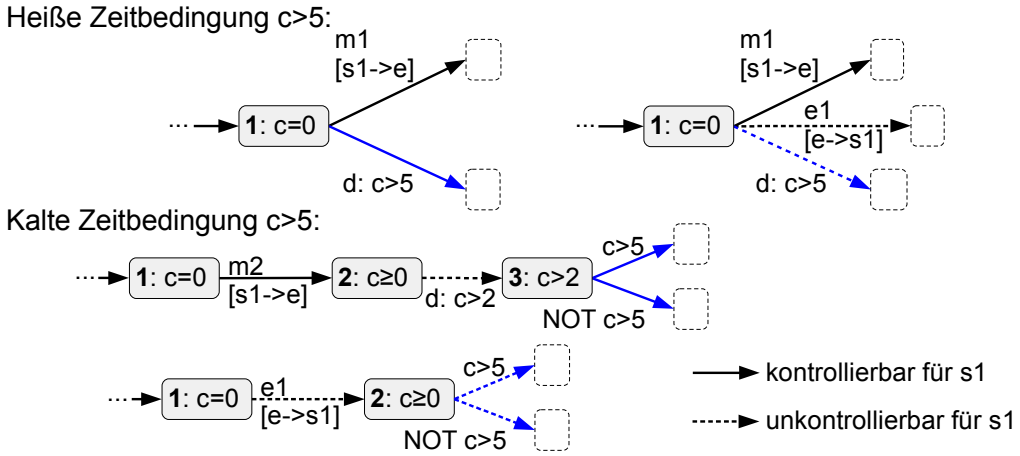


Abbildung 4.5: Beispiele für die Kontrollierbarkeit von Zeitereignissen

Ein Zeitentscheidungsereignis für eine kalte Zeitbedingung `c > 5` ist dann kontrollierbar, wenn es vor dem SG-Zustand, in dem die Bedingung erreicht wird, und nach dem letzten Clock Reset von `c` eine kontrollierbare Transition gibt. Dies ist für das obere der beiden Beispiele mit kalter Bedingung in Abbildung 4.5 der Fall, obwohl dazwischen ein Verzögerungsereignis auftritt. Für das untere Beispiel gilt diese Voraussetzung jedoch nicht: Hier bestimmt die Umgebung durch den Sendezeitpunkt von $e1$, ob `c > 5`

anschließend erfüllt ist oder nicht. Hier sind die Zeitentscheidungsereignisse daher unkontrollierbar.

Verhinderbarkeit

Bei zeitbehafteten Systemen ist die bisherige Annahme unrealistisch, dass das Senden einer Nachricht durch einen Controller in einem SG-Zustand das Auftreten aller Umgebungsereignisse verhindert: Die *Schaltzeiträume*, in denen ein Controller Transitionen tatsächlich schalten darf, können durch die im nächsten Abschnitt eingeführten *unsicheren Zones* eingeschränkt werden. Dies sind Zeitbereiche, in denen das System die Anforderungen verletzt. Eine kontrollierbare Transition ist also nur zu Zeitpunkten *schaltbar*, zu denen dies nicht zum Erreichen einer unsicheren Zone führt. Analog dazu können auch die Schaltzeiträume unkontrollierbarer Transitionen eingeschränkt sein, wenn sonst Umgebungsannahmen verletzt werden. Sind alle kontrollierbaren Transitionen erst mit Verzögerung schaltbar, so können zuvor unkontrollierbare Transitionen schalten.

Konkret kann ein Controller das Schalten einer unkontrollierbaren Transition nur durch das frühere Schalten einer für diesen Controller kontrollierbaren Transition verhindern. Diese Transition muss dazu spätestens zum gleichen Zeitpunkt wie die unkontrollierbare Transition schaltbar sein – unabhängig davon, mit welchen Uhrenwerten der initialen Zone der Zustand betreten wird. Ist dem Zustand also eine initiale Zone zugeordnet, gemäß der ein Betreten des Zustands nach Ablauf der Schaltzeiträume aller kontrollierbaren Transitionen des Controllers möglich ist, dann kann der Controller das Schalten von später (noch) schaltbaren unkontrollierbaren Transitionen nicht verhindern.

Definition 10 (Verhinderbarkeit unkontrollierbarer Transitionen)**.** Eine unkontrollierbare SG-Transition t_u ist *verhinderbar durch einen Controller*, wenn für diesen implementierte kontrollierbare ausgehende Transitionen T_c im Ausgangszustand von t_u existieren, sodass

- der Schaltzeitraum von t_u nicht vor dem frühesten Schaltzeitraum aller T_c beginnt,
- der Schaltzeitraum von t_u abzüglich der Schaltzeiträume aller T_c keine Schnittmenge mit der initialen Zone des SG-Zustands hat und
- es keinen Bereich der initialen Zone nach allen T_c, aber vor t_u gibt.

In Fällen, in denen keine Zeitbedingungen modelliert wurden, soll die zeitbehaftete Synthese dieselben Ergebnisse liefern wie die nicht zeitbehaftete Variante. Dort wird angenommen, dass das System immer vor der Umgebung schalten kann („synchrony hypothesis“, siehe Abschnitt 2.1.6). Im zeitbehafteten Fall wird daher bei *gleichzeitig* schaltbaren kontrollierbaren und unkontrollierbaren Transitionen immer angenommen, dass eine der kontrollierbaren Transitionen zuerst schaltet und damit ein Schalten der unkontrollierbaren Transitionen verhindert. Die betreffende kontrollierbare Transition muss dann allerdings spätestens zu dem Zeitpunkt schalten, zu dem die erste unkontrollierbare Transition schaltbar wird.

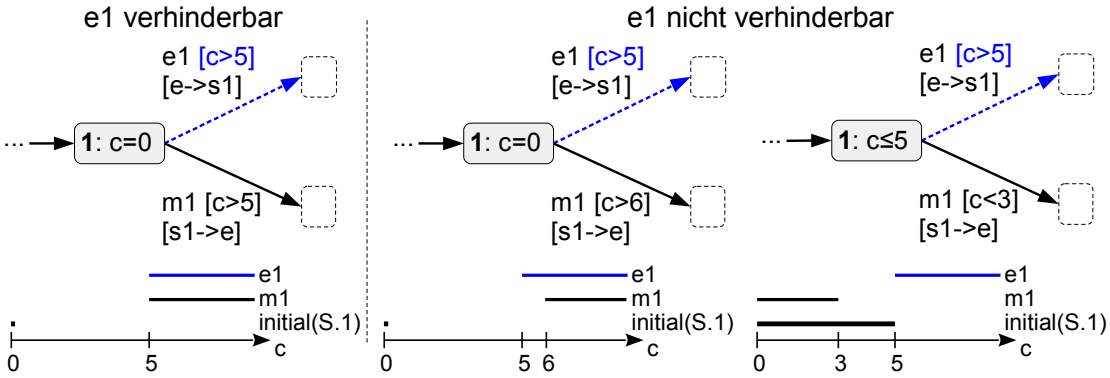


Abbildung 4.6: Beispiele für die Verhinderbarkeit von unkontrollierbaren Transitionen

Abbildung 4.6 zeigt Beispiele für Transitionen mit dem Ereignis `e1`, die für `s1` verhinderbar bzw. nicht verhinderbar sind. Im Beispiel links ist `e1` verhinderbar, da die Transition nur dann schalten kann, wenn auch die kontrollierbare Transition mit dem Ereignis `m1` schaltbar ist, die dann Vorrang hat. In den anderen beiden Fällen ist `e1` nicht verhinderbar. Im mittleren Beispiel ist dies der Fall, da `e1` bereits vor `m1` auftreten kann. Im rechten Beispiel kann `m1` zwar vor `e1` auftreten, aber Zustand S.1 kann mit $c \geq 3$ betreten werden, wodurch `m1` nicht mehr auftreten kann, `e1` aber durchaus.

4.3 Ein Algorithmus für verteilte zeitbehaftete Synthese

Dieser Abschnitt beschreibt die zeitbehafteten Erweiterungen des verteilten Syntheseverfahrens, die zusätzlich zu der in Abschnitt 4.2 beschriebenen Erweiterung des Spezifikationsgraphen (SG) erforderlich sind.

Grundsätzlich ist die Echtzeitvariante des Syntheseverfahrens eine Erweiterung des in Kapitel 3 beschriebenen Verfahrens und basiert auf derselben Grundidee: Ein Kandidat für ein Controllersystem, der zunächst keinerlei Verhalten definiert, wird schrittweise durch Hinzufügen von Transitionen zu den einzelnen Controllern zu neuen Kandidaten erweitert. Die erzeugten Kandidaten werden in einem Kandidatengraph verwaltet. Wenn ein Kandidat zu keiner Lösung mehr erweiterbar ist, wird dieser verworfen und basierend auf einem früheren Kandidaten wird ein neuer Kandidat erzeugt. Ein gegebener Kandidat ist eine Lösung (des Problems der verteilten Synthese), wenn es für ihn keine erreichbaren unimplementierten SG-Zustände gibt. Um festzustellen, wann ein SG-Zustand durch die Controller implementiert ist, wird modelliert, welche lokalen Controllerzustände welchen globalen Zuständen des SG entsprechen.

Durch die Erweiterungen in Abschnitt 4.2 kann der zeitbehaftete SG das global erlaubte Verhalten repräsentieren, das durch eine zeitbehaftete MSD-Spezifikation definiert wird. Ohne Änderungen kann der bisher beschriebene Synthesealgorithmus jedoch auch auf dieser Grundlage noch keine zeitbehaftete Implementierung erzeugen. Insbesondere muss die Definition einer Gewinnstrategie (siehe Abschnitt 3.2.2) angepasst werden, was auch entsprechende Anpassungen des Algorithmus erfordert.

Der Algorithmus muss in der zeitbehafteten Variante so erweitert werden, dass er auch Zeitereignisse korrekt behandelt. Die Betrachtung des aktuellen Zustands reicht dazu nicht aus: Um zu vermeiden, dass ein Zeitereignis auftritt, welches einer Verletzung der Anforderungen entspricht, muss das System möglicherweise bereits vor diesem Ereignis *in den richtigen Zeiträumen* Nachrichten senden. In welchen Zeiträumen Nachrichten gesendet werden dürfen und in welchen nicht, muss deshalb für alle Zustände im SG ermittelt werden, von denen aus das problematische Zeitereignis potentiell erreicht werden kann. Der Synthesealgorithmus modelliert diese Informationen durch spezielle Clock Zones in allen SG-Zuständen für das *zeitabhängig unsichere* Verhalten des Systems.

Im Folgenden beschreibt zunächst Abschnitt 4.3.1 die Unterschiede des Problems der zeitbehafteten verteilten Synthese zu der nicht zeitbehafteten Variante und definiert die Lösungen dieses Problems. Abschnitt 4.3.2 erläutert abstrakt, wie das oben erwähnte zeitabhängig unsichere Verhalten modelliert werden kann. Darauf aufbauend werden in Abschnitt 4.3.3 die erforderlichen Änderungen des Synthesealgorithmus im Einzelnen erläutert. Auch der so erweiterte Synthesealgorithmus erzeugt jedoch noch keine fertigen zeitbehafteten Controller: Diese müssen in einem weiteren Schritt, der in Abschnitt 4.3.4 behandelt wird, aus dem Syntheseergebnis extrahiert werden. In Abschnitt 4.3.5 wird informell argumentiert, dass auch das zeitbehaftete Syntheseverfahren korrekt ist. Schließlich illustriert Abschnitt 4.3.6 das Verfahren durch eine Ausführung für das zeitbehaftete Beispiel aus Abschnitt 4.1.

4.3.1 Problem und Lösungen der zeitbehafteten verteilten Synthese

In diesem Abschnitt wird zunächst kurz die zeitbehaftete Variante des Problems der verteilten Synthese von der nicht zeitbehafteten abgegrenzt. Anschließend wird auch die Definition einer Lösung des Problems an Echtzeitsysteme angepasst.

Das Problem der zeitbehafteten verteilten Synthese

Das Problem der *zeitbehafteten verteilten Synthese* besteht darin, Verhaltensmodelle für eine gegebene Menge an Systemobjekten so zu erzeugen, dass diese eine gegebene *zeitbehaftete* MSD-Spezifikation implementieren (siehe Abschnitt 2.1.6). Es unterscheidet sich von dem in Abschnitt 3.2.2 beschriebenen Problem der (nicht zeitbehafteten) verteilten Synthese also nur darin, dass die Spezifikation Echtzeitannahmen und -anforderungen enthalten kann. Daraus ergibt sich, dass eine Implementierung die Spezifikation möglicherweise nur dann erfüllen kann, wenn sie nicht nur die richtigen Nachrichten sendet, sondern dies auch in den richtigen Zeitintervallen tut.

Im Allgemeinen ist eine *Lösung* der zeitbehafteten verteilten Synthese also ein Controllersystem aus *zeitbehafteten Controllern* (siehe Abschnitt 2.2.2) das die Einhaltung der zeitbehafteten Spezifikation garantiert. Wie im nicht zeitbehafteten Fall, werden auch hier offene Systeme mit Umgebungsannahmen betrachtet. Diese Annahmen können auch zeitabhängige Einschränkungen des Umgebungsverhaltens definieren, da auch diese mit Timed MSDs definiert werden.

Analog zum nicht zeitbehafteten Fall (vgl. Abschnitt 3.2.2) müssen im Unterschied zur zeitbehafteten *globalen* Synthese die Systemobjekte autonom aufgrund lokaler In-

formationen entscheiden. Wie bereits in Abschnitt 4.2.4 diskutiert wurde, sind auch Zeitereignisse nicht für alle Objekte beobachtbar. Auch hierdurch können zusätzliche Synchronisationsnachrichten zum Austausch von Informationen zwischen den Objekten notwendig werden. Umgekehrt können durch beobachtbare Zeitereignisse aber auch andernfalls erforderliche Synchronisationsnachrichten überflüssig werden.

Werden die Übertragungszeiten für Nachrichten berücksichtigt, so kann das Hinzufügen zusätzlich benötigter Kommunikation ohne Verletzung der Anforderungen unmöglich sein. Dadurch kann das Problem für eine gegebene Spezifikation unlösbar sein, für die es ohne Berücksichtigung der Übertragungszeiten lösbar ist (vgl. Abschnitt 4.4).

Lösungen und Lösungskandidaten

In Abschnitt 3.2.2 wurde eine Lösung des (nicht zeitbehafteten) verteilten Syntheseproblems als ein Controllersystem definiert, das einer Gewinnstrategie entspricht. Die zugrunde liegende Definition einer Gewinnstrategie aus demselben Abschnitt muss für zeitbehaftete Systeme angepasst werden, um auch die Einhaltung von Zeitbedingungen zu garantieren. Dazu wird die Definition einer Strategie einerseits so angepasst, dass diese nicht nur die zu schaltenden kontrollierbaren Transitionen auswählt, sondern auch, in welchen Zeitintervallen dieses Schalten geschieht. Andererseits muss die neue Definition unkontrollierbare Transitionen in bestimmten Fällen auch in SG-Zuständen berücksichtigen, in denen bereits ausgehende kontrollierbare Transitionen Teil der Strategie sind – was im nicht zeitbehafteten Fall nicht erforderlich ist.

Definition 11 (Zeitbehaftete Gewinnstrategie, -bedingungen)**.** Eine *zeitbehaftete Strategie* besteht aus einer Teilmenge von Transitionen eines SG und einer Abbildung dieser Transitionen auf je eine Clock Zone, den *Schaltzeitraum* der Transition. Eine zeitbehaftete Strategie ist *gewinnend* (und eine *zeitbehaftete Gewinnstrategie*), wenn die folgenden beiden *zeitbehafteten Gewinnbedingungen* gelten (vgl. Abschnitt 3.2.2):

g1$_t$ Wenn nur die Transitionen der zeitbehafteten Strategie in ihren jeweiligen Schaltzeiträumen verwendet werden, dann enthält der SG keine Deadlocks und keine Zyklen ohne Zielzustände.

g2$_t$ Die zeitbehaftete Strategie muss alle nicht verhinderbaren unkontrollierbaren Transitionen enthalten, deren Ausgangszustände über Transitionen der Strategie erreichbar sind. Die Schaltzeiträume dieser Transitionen müssen ein Schalten zu beliebigen Zeitpunkten erlauben.

Als Deadlocks gelten hier alle Zustände, die keine ausgehenden Transitionen mit nichtleerem Schaltzeitraum enthalten, mit Ausnahme von Zielzuständen ohne unkontrollierbare ausgehende Transitionen. Im Unterschied zum nicht zeitbehafteten Fall kann der SG hier auch Transitionen mit Zeitereignissen enthalten.

Die Definitionen aus Abschnitt 3.2.2 für eine Lösung bzw. einen Lösungskandidaten gelten für zeitbehaftete Systeme entsprechend, d. h. mit Bezug auf zeitbehaftete Strategien anstelle von Strategien nach der ursprünglichen Definition.

Da in zeitbehafteten Systemen nicht verhinderbare unkontrollierbare Ereignisse auch bei implementierten (in der Strategie enthaltenen) kontrollierbaren Ereignissen im gleichen Zustand betrachtet werden müssen, muss die Definition der Implementiertheit eines Zustands angepasst werden: In zeitbehafteten Systemen gilt ein SG-Zustand als *implementiert*, wenn er nach den Kriterien für nicht zeitbehaftete Systeme implementiert ist *und* in ihm alle nicht verhinderbaren unkontrollierbaren Ereignisse implementiert sind.

4.3.2 Modellierung von zeitabhängig unsicherem Verhalten

Der Hauptunterschied der zeitbehafteten verteilten Synthese zur nicht zeitbehafteten ist, dass erstere ermitteln muss, in welchen Zeitintervallen die Controller senden dürfen oder müssen, um die Zeitanforderungen einzuhalten. Sie erzeugt dann Controller, welche nur so Nachrichten senden, dass sie die problematischen Zeitbereiche vermeiden. Es kann sich aber auch herausstellen, dass dies bereits ab dem Startzustand des SG nicht möglich ist und eine korrekte Implementierung somit nicht erzeugt werden kann.

Unsichere Zones

Kann bei der nicht zeitbehafteten Synthese von einem SG-Zustand aus die Erreichbarkeit eines Zielzustands nicht garantiert werden, dann muss die Erreichbarkeit dieses SG-Zustands ausgeschlossen werden. Bei der zeitbehafteten Synthese kann es aber auch vorkommen, dass die Erreichbarkeit eines Zielzustands von den Uhrenwerten abhängt: Wird ein SG-Zustand mit den „falschen“ Werten erreicht, dann kann später eine Verletzung von Zeitbedingungen unabwendbar sein. Die Synthese muss dann eine Erreichbarkeit des Zustands mit diesen Uhrenwerten verhindern, während er für andere Werte erreichbar bleiben darf. Dies kann auch für die Vorgängerzustände bedeuten, dass ihre Erreichbarkeit entsprechend eingeschränkt werden muss. Diese Zeitbereiche müssen also im Allgemeinen rekursiv für die jeweiligen Vorgänger berechnet werden – bis das System durch passende Einschränkung des Sendeintervalls von Systemnachrichten verhindern kann, dass einer seiner Folgezustände mit problematischen Uhrenwerten erreicht wird.

Um zu modellieren, welche Werte die Uhren in einem SG-Zustand für das aktuelle Controllersystem *nicht* annehmen dürfen, verwaltet die zeitbehaftete Synthese für jeden SG-Zustand eine Menge von Clock Zones, die hier als *unsichere Zones* bezeichnet werden. Diese Zones repräsentieren Zeitbereiche, bei deren Erreichen nicht (mehr) die Erreichbarkeit eines Zielzustands vom aktuellen Zustand aus garantiert werden kann. Dies kann entweder daran liegen, dass eine Verletzung der Anforderungen dann unausweichlich ist, oder daran, dass der betreffende Bereich des SG noch nicht untersucht wurde, also nicht genügend Informationen vorliegen. Eine Einhaltung der Spezifikation kann in beiden Fällen nur garantiert werden, wenn die Uhrenwerte im System niemals in einer dieser Zones liegen. Für die unsicheren Zones $unsafe(s)$ eines SG-Zustands s lassen sich durch $initial(s)^{\uparrow} - unsafe(s)$ die *sicheren Zones* des Zustands berechnen. Nach Abschluss der Synthese entsprechen diese den Schaltzeiträumen der ausgehenden Transitionen in einer zeitbehafteten Gewinnstrategie (siehe Abschnitt 4.3.1).

Um zu bewirken, dass ein SG-Zustand nur in sicheren Zeitbereichen betreten wird, kann das System gezwungen sein, nur in bestimmten Zeitbereichen Nachrichten zu sen-

den. Abbildung 4.7 zeigt ein Beispiel für einen Fall, in dem das System eine Nachricht (hier `m1`) nur bis zu einem bestimmten Zeitpunkt (hier bis maximal 5 Sekunden nach der Nachricht `e1`) senden darf, da andernfalls zwangsläufig die nachfolgende Zeitbedingung verletzt wird. Wird der Zustand S.2 hier nämlich später betreten, so tritt das Zeitentscheidungsereignis `NOT c` $\leq$ `5` auf, was eine Verletzung der heißen Zeitbedingung `c` $\leq$ `5` und damit eine Verletzung der Spezifikation darstellt. Für Zustand S.2 ist daher eine unsichere Zone `u: c > 5` definiert, da ein Betreten des Zustands mit diesen Uhrenwerten zwangsläufig zu einer Verletzung der Anforderungen führt. Diese unsichere Zone wird dann zum Zustand S.1 propagiert, da bereits in diesem ein längeres Verweilen als bis `c` $=$ `5` zwangsläufig zum Erreichen der unsicheren Zone in S.2 führt.

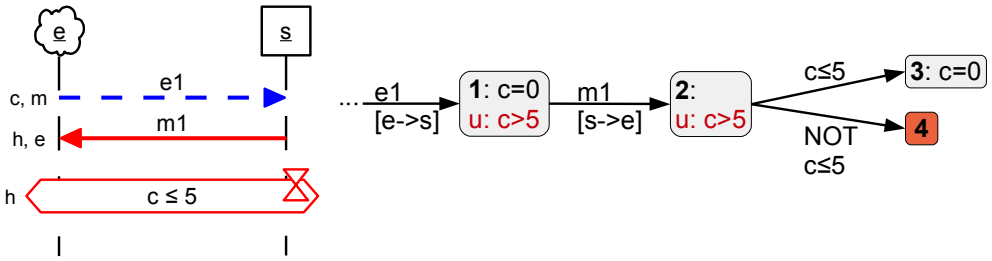


Abbildung 4.7: Beispiel für eine Nachricht, die nur in einem bestimmten Intervall gesendet werden darf

Die Erfüllung von Zeitbedingungen ist nicht immer so direkt vom Sendezeitpunkt einer Nachricht abhängig, wie im einfachen Beispiel in Abbildung 4.7. Durch die Anforderung **R7** (siehe Abbildung 4.1) des Anwendungsbeispiels in Abschnitt 4.1 müssen beispielsweise mehrere Systemnachrichten früh genug gesendet werden. Neben den beiden in diesem MSD enthaltenen Nachrichten betrifft das noch weitere Nachrichten – aufgrund der Abhängigkeiten, die durch die nicht zeitbehafteten MSDs definiert werden. Zudem hängt die Erfüllung der Zeitbedingung in **R7** auch von der Umgebung ab. Diese muss zwar aufgrund der Annahme **A3** (siehe Abbildung 4.2) nach einer bestimmten Zeit senden. Davor darf sie jedoch warten. Diese Wartezeit muss von der Synthese berücksichtigt werden, da sie die erlaubte Wartezeit des Systems entsprechend einschränkt.

Abbildung 4.8 zeigt den Zusammenhang der Exploration des zeitbehafteten SG und der Rückpropagierung der unsicheren Zones: Wie bei der nicht zeitbehafteten Synthese wird der SG so konstruiert bzw. erweitert, dass er zumindest diejenigen SG-Zustände enthält, die für den aktuellen Controllersystem-Kandidaten erreichbar sind. Zunächst werden alle erreichten SG-Zustände als sicher angenommen. Im Beispiel wird durch Implementieren eines Ereignisses eine Zeitbedingung erreicht, deren Verletzung eine Verletzung der Spezifikation darstellt. Da die Bedingung für die aktuell möglichen Uhrenwerte bei ihrem Erreichen sowohl erfüllt als auch verletzt sein kann, wird sie im SG durch einen Zeitentscheidungszustand mit zwei ausgehenden Transitionen für das Erfüllen bzw. Verletzen der Bedingung repräsentiert. Damit die Transition für die verletzte Bedingung nicht schaltbar ist, wird im Zeitentscheidungszustand eine unsichere Zone erzeugt.

Dies wiederum macht eine Berechnung der unsicheren Zones der Vorgängerzustände erforderlich. Im Beispiel wird für einen der Vorgänger eine unsichere Zone erzeugt, für

dessen Vorgänger jedoch nicht, womit die Rückpropagierung endet. Diese Rückpropagierung wird immer dann ausgeführt, wenn bei der Erzeugung eines neuen Kandidaten der SG so erweitert wird, dass für mindestens einen SG-Zustand neue unsichere Zones erreichbar werden.

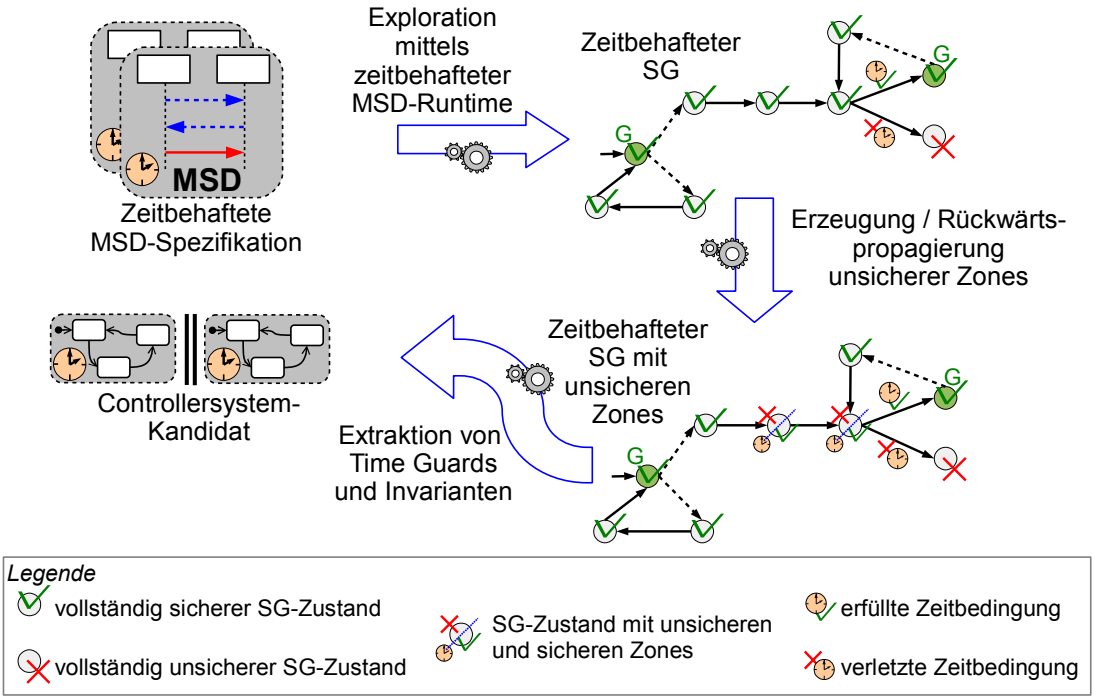


Abbildung 4.8: Überblick über die zur Durchführung der verteilten Synthese erforderlichen Schritte

Mittels der unsicheren Zones kann nach erfolgreicher Synthese schließlich bestimmt werden, wie die eingehenden kontrollierbaren Transitionen durch Time Guards und Invarianten in den Controllern eingeschränkt werden müssen, um ein Erreichen eines Zustands mit Uhrenwerten in einer unsicheren Zone zu vermeiden. Dies wird hier als „Extraktion“ bezeichnet.

Die Idee, unsichere Clock Zones für die einzelnen SG-Zustände zu verwalten, basiert auf einem ähnlichen Vorgehen der in Uppaal Tiga umgesetzten Strategiesuche für Timed Game Automata [CDF+05], bzw. auf dem früheren Verfahren von Maler et al. [MPS95]. Ziel dieser Strategiesuche ist die Ermittlung der Transitionen und Schaltzeiträume, die einem Spieler ein Gewinnen des modellierten Spiels garantieren – es werden dort also sichere statt unsichere Zones betrachtet. Zusätzlich wird für die Strategiesuche in Uppaal Tiga als Optimierung des Verfahrens vorgeschlagen, auch die Clock Zones zu ermitteln, in denen ein Verlieren unausweichlich ist. Die in dieser Arbeit definierten unsicheren Zones werden jedoch nicht nur für Situationen eines unausweichlichen Verlierens verwendet, sondern auch für Fälle, in denen ein Erreichen solcher Situationen zwar nicht zwangsläufig ist, aber auch nicht ausgeschlossen werden kann.

Unsichere Zones werden erzeugt, wenn in implementierten Zuständen Transitionen für Zeitereignisse *nicht* implementiert werden. Dadurch entscheidet der Algorithmus, dass ein Controller zu einem früheren Zeitpunkt eine Nachricht so gesendet hat, dass das Zeitintervall für dieses Zeitereignis im betreffenden Zustand *nicht* erreichbar ist. Im Beispiel in Abbildung 4.7 entsteht eine unsichere Zone in Zustand S.2, wenn die Transition zum Zustand S.4 nicht implementiert ist (was sie hier aufgrund der sonst zwangsläufigen Verletzung der Anforderungen nicht sein darf). Die unsicheren Zones werden dann berechnet, um festzustellen, wann genau das Senden passieren musste. Dazu wird zunächst im SG-Zustand mit dem nicht implementierten Zeitereignis eine unsichere Zone identisch zur Zone des Ereignisses angelegt. Oft ist schon in den Vorgängerzuständen ein Erreichen dieser unsicheren Zone in bestimmten Zeitbereichen zwangsläufig. In diesen Vorgängerzuständen müssen dann ebenfalls unsichere Zones angelegt werden.

Unsichere Zones müssen für alle Zeitbereiche angelegt werden, für die das System das Erreichen unsicherer Zones in einem Folgezustand nicht mehr verhindern kann. Bei kontrollierbaren Transitionen kann das System das Erreichen dieser unsicheren Zones oft durch Warten bzw. durch den passenden Sendezeitpunkt einer Nachricht vermeiden. In diesen Fällen entsteht im aktuellen Zustand also keine unsichere Zone. Ist dies nicht möglich oder ist die Transition unkontrollierbar, so muss die unsichere Zone des Folgezustands übernommen werden. Dies kann bei den unsicheren Zones der (direkten oder indirekten) Vorgänger des aktuellen Zustands weitere Änderungen bedingen, durch Zyklen im SG auch mehrfach für denselben Zustand. Der Algorithmus propagiert diese Änderungen immer dann zu den Vorgängern eines Zustands weiter, wenn sich die Menge von unsicheren Zones für diesen Zustand geändert hat.

Unsichere Zones führen dazu, dass Controller in manchen SG-Zuständen nicht zu jedem Zeitpunkt schalten dürfen. Dadurch wiederum wird es möglich, dass die Umgebung Nachrichten senden kann, die sonst durch das frühere Senden eines Controllers verhindert würden. Diese Nachrichten müssen nun durch das Controllersystem und damit durch die Synthese berücksichtigt werden. Die Kriterien für die Implementiertheit von SG-Zuständen sind aus diesem Grund leicht verändert, wie in Abschnitt 4.3.1 beschrieben ist. Damit können Erweiterungen von Kandidaten, durch die unsichere Zones entstehen oder erweitert werden, dazu führen, dass bereits implementierte SG-Zustände erneut unimplementiert werden. Diese müssen dann erneut implementiert werden.

Verwendung der unsicheren Zones am Ende der Synthese

Wenn für ein gegebenes Controllersystem jeder erreichbare Zustand schließlich implementiert ist und ein Betreten des Zustands in einer unsicheren Zone ausgeschlossen ist, dann entspricht dieses Controllersystem einer Lösung des zeitbehafteten Syntheseproblems (siehe Abschnitt 4.3.1). Die erzeugten Controller enthalten jedoch noch keine zeitbehafteten Einschränkungen des Verhaltens, da die unsicheren Zones zunächst nur für die SG-Zustände berechnet werden. Die Einschränkungen des Controllerverhaltens werden durch den Synthesealgorithmus in einem nachgelagerten Schritt aus den unsicheren Zones der SG-Zustände ermittelt: Das Schalten aller kontrollierbaren Transitionen muss dazu so eingeschränkt werden, dass keine unsicheren Zones des Folgezustands erreicht werden können. Dies wird durch Hinzufügen von Time Guards zur entsprechenden

Transition des sendenden Controllers ermöglicht, die ein Schalten nur dann erlauben, wenn dies nicht zum Erreichen einer unsicheren Zone im Folgezustand führt.

Wenn im aktuellen SG-Zustand unsichere Zones existieren, die allein durch Warten erreicht werden können, dann muss zusätzlich ein rechtzeitiges Verlassen des Zustands garantiert werden. Dies wird durch Erzeugen geeigneter Invarianten im Ausgangszustand der sendenden Transition des Controllers erzielt.

Auf dieselbe Weise wird garantiert, dass jeder Zustand, der kein Zielzustand ist, irgendwann verlassen wird. Ohne Invariante könnte der Controller abwarten bis der letzte mögliche Schaltzeitpunkt aller Transitionen vorbei ist und somit kein Verlassen des Zustands mehr möglich ist. Dies würde einem endlosen Verweilen eines Requirement MSDs in einem ausgeführten Cut entsprechen, also der Verletzung einer Liveness-Anforderung.

Ein Controller darf vor dem Senden einer Nachricht generell nur so lange warten, wie er dadurch keine unsichere Zone erreicht. Da die unsicheren Zones sich während der Synthese noch ändern können, kann die endgültig mögliche Wartezeit also erst nach der Terminierung bestimmt werden. Aus den gegebenen unsicheren Zones kann dann die mögliche Wartezeit in Abhängigkeit von den betreffenden Uhren bestimmt werden. Der Controller kann die verbleibende erlaubte Wartezeit jedoch nur dann ermitteln, wenn er im aktuellen Zustand alle Uhren beobachten kann, die im Zustand Werte innerhalb und außerhalb einer verbotenen Zone haben. Der Wert dieser Uhren entscheidet dann darüber, ob die jeweilige unsichere Zone erreicht ist. Ohne Kenntnis dieser Uhrenwerte kann der Controller nicht bestimmen, wann er zu lange gewartet hat und muss daher sofort senden.

Die mögliche Wartezeit eines Controllers schränkt sich insbesondere in Fällen ein, in denen das Feuern von unkontrollierbaren Transitionen durch eine kontrollierbare Transition verhindert wurde. Das Schalten der betreffenden kontrollierbaren Transition darf dann nur maximal bis zum Beginn des frühesten Schaltzeitraums der unkontrollierbaren Transitionen verzögert werden. Auch hier gilt: Wenn die betreffenden Uhren nicht durch den Controller beobachtbar sind, muss das Schalten der kontrollierbaren Transition (also das Senden der entsprechenden Nachricht durch den Controller) sofort erfolgen. Die „betreffenden“ Uhren sind diejenigen, die im aktuellen Zustand sowohl Werte innerhalb als auch außerhalb des Schaltzeitraums der unkontrollierbaren Transitionen haben können.

Wir bezeichnen ein Nachrichtenereignis als *verzögerbar*, wenn es durch einen Controller gesendet wird, dieser die Uhren aller unsicheren Zones im aktuellen SG-Zustand beobachten kann und wenn nicht bereits bei minimaler Wartezeit eine unsichere Zone erreicht wird.

4.3.3 Erweiterung des Algorithmus für verteilte Synthese

Der in Abschnitt 3.3 beschriebene Algorithmus muss für den zeitbehafteten Fall teilweise angepasst werden, wie im vorhergehenden Abschnitt bereits erläutert wurde. Insbesondere erfordert die zeitbehaftete Definition einer Lösung in Abschnitt 4.3.1 Anpassungen. Weiterhin müssen für die Zustände des SG, die für einen gegebenen Lösungskandidaten erreichbar sind, zusätzlich zur initialen Zone die in Abschnitt 4.3.2 eingeführten unsicheren Zones berechnet werden.

Neben unsicheren Zones berechnet der Algorithmus zudem die ähnlichen *ausgeschlossenen Zones*, die zeitbehaftetes Umgebungsverhalten repräsentieren, welches zwangsläufig die Annahmen verletzt. Ausgeschlossene Zones modellieren für SG-Zustände Uhrenwerte, bei denen sich das System für eine zu den Annahmen konformen Umgebung nicht im betreffenden Zustand aufhalten kann.

Die Erzeugung neuer Kandidaten ändert sich nicht grundsätzlich, sie berücksichtigt jetzt aber auch Zeitereignisse. Insbesondere müssen die erzeugten Kandidaten diese neuen Ereignisse nun auch implementieren. Um einen Kandidaten so zu erweitern, dass er ein Zeitereignis implementiert, wird in jedem Controller, der das Ereignis beobachten kann, eine Transition zu einem neuen Zustand hinzugefügt, die mit dem Ereignis annotiert ist. Der betreffende Controller kann dadurch die SG-Zustände vor dem Ereignis von denen nach dem Ereignis unterscheiden. Im Folgezustand der neuen Transition kann er dann eine passende Reaktion auf das Ereignis definieren.

Wie im nicht zeitbehafteten Fall werden anschließend durch eine Tiefensuche für den neuen Controllerzustand die Korrespondenzen zu SG-Zuständen ermittelt. Diese Tiefensuche muss im zeitbehafteten Fall auch für Zeitereignisse berücksichtigen, ob sie beobachtbar sind oder nicht. Durch die erweiterten Definitionen für Beobachtbarkeit in Abschnitt 4.2.4 wird das aber bereits berücksichtigt. Auch die Kontrollierbarkeit dieser Ereignisse wird entsprechend der neuen Definition überprüft.

Die Ermittlung erreichbarer unimplementierter Zustände funktioniert wie bisher, jedoch nach den beschriebenen veränderten Kriterien. Bei Prüfung der Implementiertheit muss nun die neue Definition (siehe Abschnitt 4.3.1) verwendet werden. Diese erfordert die Information, ob die unkontrollierbaren Nachrichtenereignisse durch das System verhinderbar sind oder nicht. Das wird nach den in Abschnitt 4.2.4 aufgeführten Kriterien überprüft. Diese Kriterien beziehen sich auf die Schaltzeiträume der Transitionen, die sich durch die nachfolgend genauer definierten unsicheren und ausgeschlossenen Zones ergeben. Sie werden am Ende dieses Abschnitts konkretisiert. Da nicht verhinderbare unkontrollierbare Nachrichtenereignisse in jedem Fall implementiert werden müssen, werden sie bei der Auswahl der zu implementierenden Ereignisse mit Vorrang vor kontrollierbaren Ereignissen berücksichtigt – im Gegensatz zu verhinderbaren unkontrollierbaren Ereignissen.

Weiterhin muss nun immer dann, wenn ein SG-Zustand implementiert wurde, auf unimplementierte Zeitereignisse in diesem Zustand geprüft werden. Falls solche existieren, so ruft der Algorithmus die neue Prozedur updateUnsafeZones auf. Diese berechnet die unsicheren Zones des betreffenden SG-Zustands und, falls erforderlich, von dessen Vorgängerzuständen.

Beim Aufruf von updateUnsafeZones kann es vorkommen, dass für einen Zustand ermittelt wird, dass seine initiale Zone sich vollständig mit unsicheren Zones überlappt. In diesem Fall muss der betreffende Kandidat verworfen werden, da sonst ein Erreichen der unsicheren Zones unausweichlich wäre. Dazu wird der Algorithmus geringfügig angepasst, sodass er nun in allen erreichbaren Zuständen auf eine Überlappung initialer und unsicherer Zones prüft: Jeder Kandidat, für den eine solche Überlappung existiert, wird verworfen.

Nachfolgend wird zunächst die Berechnung der unsicheren Zones eines einzelnen SG-Zustands basierend auf seinen ausgehenden Transitionen beschrieben. Anschließend wird

die Propagierung unsicherer Zones zu den Vorgängerzuständen erläutert. Zum Schluss wird die ähnliche Erzeugung ausgeschlossener Zones behandelt.

Berechnung der unsicheren Zones

Unsichere Zones können in einem SG-Zustand auf zwei Arten entstehen: Zum einen können völlig neue unsichere Zones durch unimplementierte Zeitereignisse des Zustands entstehen. Zum anderen kann die Erzeugung unsicherer Zones im aktuellen Zustand aufgrund unsicherer Zones in Folgezuständen erforderlich sein, da ein Erreichen der betreffenden Zeitbereiche im aktuellen Zustand zwangsläufig zum Erreichen dieser unsicheren Zones in den Folgezuständen führen würde. In beiden Fällen sind die erzeugten unsicheren Zones des jeweils betrachteten SG-Zustands abhängig von seinen ausgehenden Transitionen.

Durch die Erzeugung von unsicheren Zones im aktuellen Zustand kann die Erzeugung unsicherer Zones auch rekursiv für die Vorgängerzustände erforderlich sein („Rückpropagierung“). Hier wird zunächst die Berechnung der unsicheren Zones für einen einzelnen Zustand, also ohne Rekursion, betrachtet. Der nachfolgende Abschnitt beschreibt die Rückpropagierung.

Die Berechnung der unsicheren Zones ist unterschiedlich für Zeitentscheidungszustände und andere SG-Zustände. In beiden Fällen müssen unsichere Zones aus Folgezuständen zur Übernahme in den aktuellen Zustand angepasst werden. Zunächst wird diese Anpassung diskutiert, dann die spezielle Berechnung für die beiden Zustandsarten.

Übernahme von unsicheren Zones aus Folgezuständen Die angepassten unsicheren Zones des über die Transition t erreichbaren Folgezustands (`target(t)`) eines SG-Zustands s können durch die nachfolgend definierte Funktion `successor_unsafe(s,t)` ermittelt werden. Die Funktion `unsafe(s)` gibt die Federation (siehe Abschnitt 2.2.2) der bisher berechneten unsicheren Zones eines SG-Zustands s an. Zur Anpassung der einzelnen Clock Zones wird die Hilfsfunktion `adapt(x,s,s')` verwendet. Diese erzeugt für eine gegebene Clock Zone x in Folgezustand s' eine für s angepasste Clock Zone. Die Funktionen sind wie folgt definiert:

$$\begin{aligned} successor\_unsafe(s,t) &= \bigvee_{z \in unsafe(target(t))} adapt(z,s,target(t)) \\ adapt(x,s,s') &= restrict\_clocks(free(x \wedge initial(s'), reset\_clocks(s')),s) \end{aligned} \tag{4.2}$$

Die Funktion `adapt` verwendet zur Anpassung der Clock Zones weitere Hilfsfunktionen. Nachfolgend wird die Berechnung der angepassten Clock Zone unter Verwendung dieser Funktionen erläutert.

Der erste Schritt der Berechnung von `adapt(x,s,s')` ist, x mit `initial(s')` zu schneiden. Dabei werden Bedingungen über Uhren, die nicht in beiden Clock Zones vorkommen, ignoriert. Diese Anpassung von x ist erforderlich, da unsichere Zones in s' oft nicht für s übernommen werden dürfen, wenn sie sich auf in s' zurückgesetzte Uhren beziehen: Wenn die Clock Zone x für eine Uhr `c`, die in s' zurückgesetzt wird, eine untere

Schranke definiert (z. B. `c > 3`) so hat x keine Entsprechung in s. Die untere Schranke bezieht sich auf den Wert von `c` *nach* dem Clock Reset und darf daher den Wert von `c` in s *vor* dem Reset nicht einschränken: Unabhängig von der in s vergangenen Zeit gilt beim Betreten von s' `c = 0`, womit x noch nicht erreicht ist. Fordert x also beispielsweise `c > 3` und in `initial(s')` gilt `c = 0`, so ist die Schnittmenge leer, wodurch `adapt(x,s,s') = false` gilt. Der Fall, dass ein Zustand durch ein Verzögerungsereignis nur zu exakt einem Zeitpunkt betreten werden kann, wird auf dieselbe Weise wie ein Clock Reset behandelt, mit dem Unterschied, dass der betreffende Uhrwert > 0 ist.

Ist die resultierende Clock Zone nicht leer, so werden aus ihr durch die Hilfsfunktion `free` alle Bedingungen entfernt, die sich auf die in s' zurückgesetzten Uhren beziehen. Dadurch werden auch die Clock Resets selbst rückgängig gemacht, da diese in s noch nicht stattgefunden haben. In der resultierenden Clock Zone werden schließlich durch `restrict_clocks` Uhren so entfernt (`clocks(s') \ clocks(s)`) bzw. hinzugefügt (`clocks(s) \ clocks(s')`) dass die Uhrenmenge mit der von s übereinstimmt. Die so erzeugt Clock Zone ist dann das Ergebnis von `adapt(x,s,s')`.

Berechnung für Zeitentscheidungszustände In Zeitentscheidungszuständen werden bei implementierten ausgehenden Transitionen die Clock Zones des Folgezustands (angepasst) übernommen und mit der Bedingung des betreffenden Zeitereignisses geschnitten. Für nicht implementierte Zeitentscheidungsereignisse wird eine unsichere Zone für den gesamten Zeitbereich angelegt, der durch diese repräsentiert wird. Davon ausgenommen sind Zeitentscheidungsereignisse, die direkt zu einer Verletzung der Annahmen führen (z. B. weil sie die Verletzung einer heißen Zeitbedingung mit oberer Schranke in einem Assumption MSD repräsentieren). Die Berechnung der unsicheren Zones von Zeitentscheidungszuständen ist daher wie folgt definiert:

In einem Zeitentscheidungszustand s mit einer Menge ausgehender Transitionen T_o ist die Federation der unsicheren Zones `unsafe(s)` definiert als:

$$unsafe(s) = \bigvee_{t \in T_o} unsafe(t) \tag{4.3}$$

Dabei ist `unsafe(t)` für eine Transition t mit Folgezustand s' und Zeitentscheidungsereignis e mit Bedingung c wie folgt definiert:

$$unsafe(t) = \begin{cases} \{c\} & \text{falls } e \text{ Annahmen-konform und unimplementiert ist und das Ereignis für } \neg\, c \text{ in } s \text{ implementiert ist} \\ successor\_unsafe(s,t) \wedge c & \text{falls } t \text{ implementiert ist} \\ \emptyset & \text{sonst} \end{cases} \tag{4.4}$$

Abbildung 4.9 zeigt vier Beispiele für die Berechnung der unsicheren Zones eines Zeitentscheidungszustands s basierend auf den ggf. vorhandenen Folgezuständen bzw. den unimplementierten Transitionen für die Zeitentscheidungsereignisse. Der Ausdruck nach den Zustandsnamen gibt jeweils die initiale Zone an (der Ausdruck fehlt, wenn diese unbeschränkt ist); die unsicheren Zones werden mit dem Präfix „u:“ gekennzeichnet.

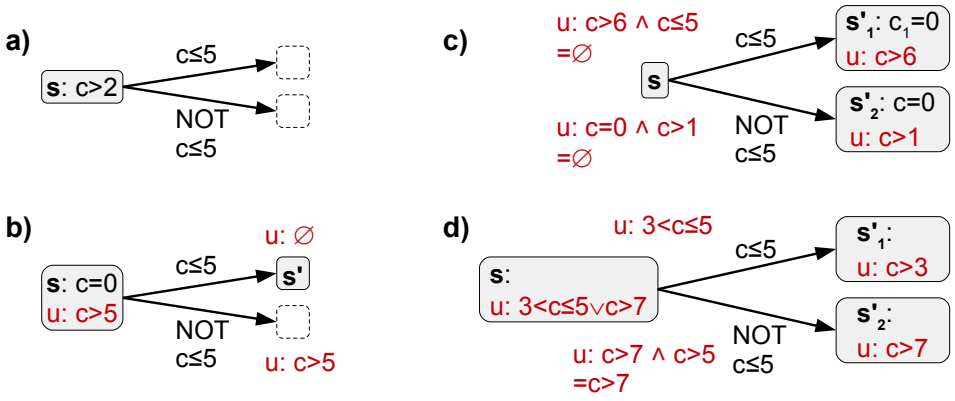


Abbildung 4.9: Beispiele für die Berechnung von unsicheren Zones für Zeitentscheidungszustände

Beispiel **a** zeigt einen Fall, in welchem beide Transitionen nicht exploriert und damit auch nicht implementiert sind: Hier ist s nach der Definition nie unsicher. In Beispiel **b** ist die erfüllte Bedingung $\texttt{c} \leq 5$ implementiert und führt zum Zustand s'ohne unsichere Zone. Daher ergibt sich die unsichere Zone von s aus der unimplementierten Transition der unerfüllten Bedingung und entspricht daher der negierten Bedingung, also $\texttt{c} > 5$.

In Beispiel **c** sind beide Transitionen implementiert und führen zu Zuständen mit unsicheren Zones. Im Falle von s'_1 schließen sich jedoch die unsichere Zone und die Zeitbedingung der Transition aus. Der Clock Reset der Uhr c_1 in s'_1 ist hier irrelevant für die unsicheren Zones, da diese Uhr in ihnen nicht verglichen wird. In s'_2 wird die Uhr c zurückgesetzt, wodurch die unsichere Zone für $\texttt{c} > 1$ beim Betreten des Zustands noch nicht erreicht ist und der Uhrwert für c in s *vor* dem Reset für die unsichere Zone irrelevant ist. Diese unsichere Zone wird daher ebenfalls nicht zu s propagiert.

In Beispiel **d** schließlich sind wieder beide Transitionen implementiert. Hier ergeben sich aufgrund der unsicheren Zones der beiden Folgezustände durch Schneiden mit der jeweiligen Bedingung zwei unsichere Zones in s.

Berechnung für normale SG-Zustände In SG-Zuständen ohne Zeitentscheidungsereignisse ergeben sich unsichere Zones in erster Linie durch die unsicheren Zones in Folgezuständen von Transitionen mit Nachrichtenereignissen. Dabei ist die Unterscheidung zwischen kontrollierbaren und unkontrollierbaren Transitionen für die Berechnung der unsicheren Zones wichtig. Wann immer die Umgebung über das Auftreten eines Ereignisses bestimmt, muss jeweils der ungünstigste Fall angenommen werden. Wenn also mehrere Ereignisse implementiert sind oder wenn Ereignisse unkontrollierbar und nicht verhinderbar sind, dann sind im aktuellen Zustand alle Zeitbereiche unsicher, die im Folgezustand für eines der betreffenden Ereignisse unsicher sind.

Entsprechend der Ereignisse, durch die die unsicheren Zones erreicht werden können, wird zwischen unkontrollierbaren und kontrollierbaren unsicheren Zones unterschieden. Zudem sind die über kontrollierbare Ereignisse erreichbaren sicheren Zones relevant.

Die betreffenden Federations werden durch die nachfolgend eingeführten Hilfsfunktionen berechnet und erst in einem zweiten Schritt zu den endgültigen unsicheren Zones kombiniert. Bei kontrollierbaren unsicheren Zones gilt dabei, dass diese nur in solchen Zeiträumen zu einer unsicheren Zone im aktuellen Zustand führen, in denen sie *alle* unsicher sind; bei unkontrollierbar unsicheren Zones reicht eine einzige.

In einem SG-Zustand *ohne* Zeitentscheidungsereignisse s mit Mengen von ausgehenden kontrollierbaren implementierten Transitionen T_c und nicht verhinderbaren unkontrollierbaren Transitionen T_u ist die Federation der unkontrollierbaren unsicheren Zones durch `uc_unsafe(s)`, die der kontrollierbaren unsicheren Zones durch `c_unsafe(s)` und die der kontrollierbaren sicheren Zones durch `c_safe(s)` definiert:

$$\begin{aligned} uc\_unsafe(s) &= \bigvee_{t \in T_u} successor\_unsafe(s,t) \\ c\_unsafe(s) &= \bigwedge_{t \in T_c} successor\_unsafe(s,t) \\ c\_safe(s) &= initial(s)^{\uparrow} - c\_unsafe(s) \end{aligned} \tag{4.5}$$

Durch den Ausdruck $initial(s)^{\uparrow} - c\_unsafe(s)$ werden dabei die Zeitbereiche in s errechnet, die nicht in `c_unsafe(s)` liegen.

Auch durch Verzögerungsereignisse können unsichere Zones erreichbar werden oder neu entstehen. Wenn ein Verzögerungsereignis nicht implementiert ist, dann ist der Zeitbereich unsicher, in dem seine Zeitbedingung erfüllt ist. Ist es aber implementiert, dann werden die unsicheren Zones des Folgezustands übernommen, soweit sie Zeitbereiche nach dem Verzögerungsereignis betreffen. Weiterhin können auch Zeitbereiche vor diesem Ereignis unsicher sein, wenn die Uhrenwerte dann bereits implizieren, dass bei Erreichen des Verzögerungsereignisses keine sicheren Zones mehr erreicht werden können. Letzteres ist insbesondere dann der Fall, wenn die erforderliche Wartezeit für das Verzögerungsereignis gleichzeitig zum Erreichen der unsicheren Zones im Folgezustand führen muss.

In einem SG-Zustand *ohne* Zeitentscheidungsereignisse s mit Transitionen mit Verzögerungsereignissen T_d werden die Federations der durch Verzögerungsereignisse erreichbaren unsicheren bzw. sicheren Zones durch `d_unsafe` bzw. `d_safe` und die Federation der vor Erreichen der Bedingung entstehenden unsicheren Zones durch `pre_d_unsafe` definiert:

$$\begin{aligned} d\_unsafe(s) &= \bigvee_{t \in T_d} (successor\_unsafe(s,t) \wedge cond(t)) \\ d\_safe(s) &= initial(s)^{\uparrow} - d\_unsafe(s) \wedge \bigwedge_{t \in T_d} cond(t) \\ pre\_d\_unsafe(s) &= initial(s)^{\uparrow} - d\_safe(s)^{\downarrow} \end{aligned} \tag{4.6}$$

Dabei ermittelt `cond(d)` die Zeitbedingung des Zeitereignisses einer Transition d. Für unimplementierte Transitionen $t \in T_d$ gelte `successor_unsafe(s,t) = true`. Durch das Schneiden mit `cond(t)` in den Berechnungen von `d_unsafe` und `d_safe` wird jeweils sichergestellt, dass nur die Zeitbereiche nach Auftreten des Ereignisses betrachtet

werden. Die Berechnung von `pre_d_unsafe` ermittelt alle Zeitbereiche in s, in denen `d_safe` nicht mehr erreicht werden kann.

Die endgültigen unsicheren Zones des betrachteten Zustands werden durch die nachfolgend definierte Funktion berechnet. Diese verwendet die obigen Hilfsfunktionen, erweitert die durch diese ermittelten unsicheren Zones aber teilweise oder schränkt sie ein. Die Definition wird anschließend schrittweise erläutert.

In einem SG-Zustand *ohne* Zeitentscheidungsereignisse s ist die Federation der unsicheren Zones `unsafe(s)` definiert als:

$$\begin{aligned} unsafe(s) \;=\; & (c\_unsafe(s) - c\_safe(s)^{\downarrow}) \;\vee\; (uc\_unsafe(s) - c\_safe(s)) \\ & \vee\; d\_unsafe(s) \;\vee\; (pre\_d\_unsafe(s) - c\_safe(s)^{\downarrow}) \\ \vee \bigvee_{u \in zones(uc\_unsafe(s))} & \left(u^{\downarrow} - \bigvee_{c \in \{c' \in zones(c\_safe(s)) \mid earlier(s,c',u)\}} c^{\downarrow} \right) \end{aligned} \tag{4.7}$$

Dabei ermittelt `zones(f)` die Menge der Clock Zones einer Federation f.

Für die kontrollierbaren unsicheren Zones in `c_unsafe(s)` gilt, dass diese nur dann auch im aktuellen Zustand unsicher sind, wenn es nach ihnen keine sicheren Zones mehr gibt. Andernfalls kann das System die entsprechenden unsicheren Zones im Folgezustand durch Warten in s bis zum Erreichen der sicheren Zones vermeiden. Die sicheren Zones in `c_safe(s)` und der Zeitraum vor diesen werden daher von `c_unsafe(s)` subtrahiert $(c\_unsafe(s) - c\_safe(s)^{\downarrow})$.

Weiterhin sind auch diejenigen Zeitbereiche unsicher, in denen nicht durch Schalten einer kontrollierbaren Transition vermieden werden kann, dass über eine unkontrollierbare Transition eine unsichere Zone `uc_unsafe(s)` erreichbar wird. Diese werden durch $uc\_unsafe(s) - c\_safe(s)$ ermittelt.

Zeitbereiche, in denen eine Transition mit einem Verzögerungsereignis zu einer unsicheren Zone im Folgezustand führt (`d_unsafe(s)`), sind ebenfalls unsicher. Zusätzlich sind auch Zeitbereiche unsicher, in denen die Uhrwerte implizieren, dass ein Auftreten eines Verzögerungsereignisses zwangsläufig zu einem Erreichen einer unsicheren Zone im Folgezustand führt (`pre_d_unsafe(s)`). Dies gilt jedoch nur dann, wenn dieses Ereignis nicht mehr durch eine kontrollierbare Transition vermieden werden kann $(pre\_d\_unsafe(s) - c\_safe(s)^{\downarrow})$.

Zusätzlich sind alle Zeitbereiche unsicher, in denen die Umgebung durch Warten eine unkontrollierbare Transition erreichen kann, die zu einer unsicheren Zone im Folgezustand führt – ohne von einer implementierten kontrollierbaren Transition unterbrochen zu werden. Für die durch `uc_unsafe(s)` ermittelten unsicheren Zones gilt daher, dass der Zeitbereich vor diesen $(u^{\downarrow})$ ebenfalls unsicher ist. Dies gilt jedoch nur im Zeitraum nach der letzten kontrollierbar sicheren Clock Zone *vor* (Funktion `earlier` aus Abschnitt 4.2.3) der unsicheren Zone, da frühere kontrollierbare Transitionen ein Erreichen der betreffenden unsicheren Zone verhindern können – wenn sie zu sicheren Zones führen (Subtraktion von $c^{\downarrow}$). Insgesamt wird diese Berechnung durch die letzte Zeile der Definition durchgeführt.

Abbildung 4.10 zeigt zwei Beispiele für die Berechnung von unsicheren Zones eines SG-Zustands s ohne Zeitentscheidungsereignisse. In Beispiel **a** führen beide kontrollierbaren

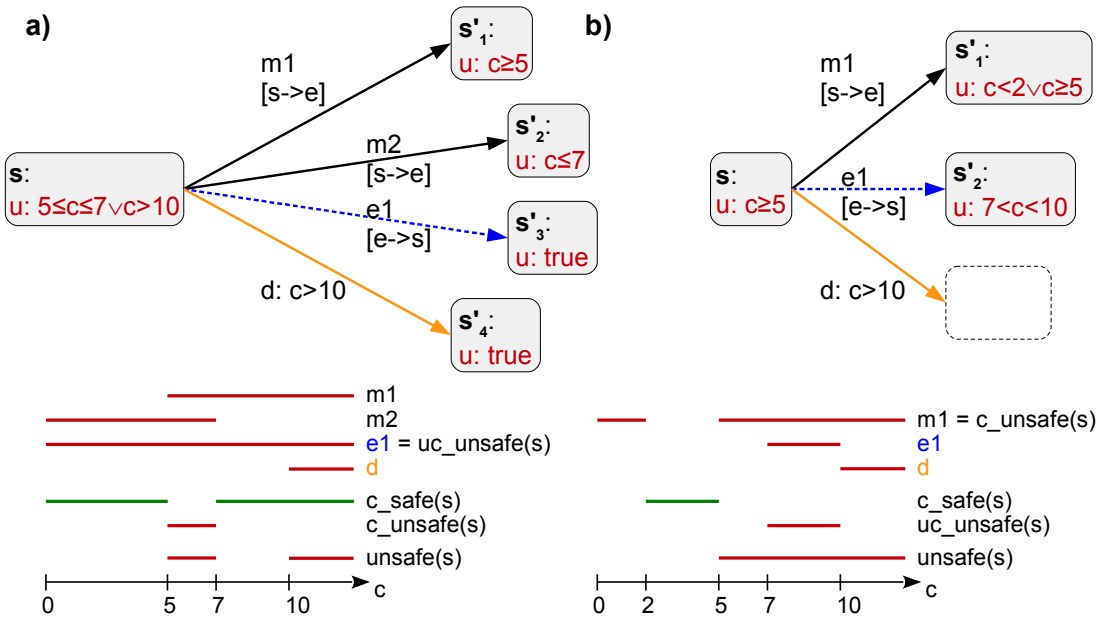


Abbildung 4.10: Beispiele für die Berechnung von unsicheren Zones für gewöhnliche SG-Zustände

Nachrichten `m1` und `m2` zu Zuständen mit unsicheren Zones. Dies gilt jedoch gleichzeitig nur im Zeitintervall $5 \leq c \leq 7$; davor und danach ist es sicher, eine der beiden Transitionen zu schalten. Durch sofortiges Schalten einer dieser Transitionen kann ein Auftreten des Umgebungsereignisses `e1` vermieden werden, dessen Folgezustand vollständig unsicher ist. Im Zeitintervall $5 \leq c \leq 7$ kann `e1` jedoch auftreten, wodurch s in diesem Intervall unsicher ist. Zudem führt ein Verzögerungsereignis zu einer unsicheren Zone im Folgezustand, wodurch auch $c > 10$ in s unsicher ist.

In Beispiel **b** ist das kontrollierbare Ereignis `m1` nur im Intervall $2 \leq c \leq 5$ sicher, da sonst eine unsichere Zone im Folgezustand betreten wird. Der Zeitbereich $c < 2$ ist in s jedoch ebenfalls sicher, da in diesem keine problematischen unkontrollierbaren Ereignisse auftreten können und das System in s bis $c \geq 2$ warten kann. Für $c \geq 5$ ist s unsicher, da ab $c = 5$ nicht mehr verhindert werden kann, dass das Umgebungsereignis `e1` durch Schalten im Intervall $7 < c < 10$ das Erreichen einer unsicheren Zone bewirkt. Bei $c > 10$ tritt ein unimplementiertes Verzögerungsereignis auf, wodurch auch dieser Zeitraum unsicher ist.

Die oben angegeben Definitionen können relativ direkt in einem imperativen Algorithmus implementiert werden. Für die Umsetzung der einfachen ein- oder zweistelligen Operationen (siehe Abschnitt 2.2.2) auf Clock Zones wurde von Bengtsson und Yi [BY04] Pseudocode veröffentlicht.

Propagierung der unsicheren Zones

Die Propagierung unsicherer Zones wird durch die Prozedur updateUnsafeZones durchgeführt. Diese wird initial immer für einen SG-Zustand aufgerufen, der gerade implementiert wurde. Zusätzlich muss gelten, dass es in diesem ein nicht implementiertes Zeitereignis gibt, oder dass dieser eine neue ausgehende Transition zu einem SG-Zustand mit unsicheren Zones hat.

Im ersten Fall entsteht eine neue unsichere Zone, die durch die iterative Definition der unsicheren Zones auch eine erneute Berechnung für alle (direkten oder indirekten) Vorgängerzustände erfordern kann. Daher müssen auch für diese Zustände die unsicheren Zones berechnet werden, was durch einen rekursiven Aufruf von updateUnsafeZones geschieht. Im zweiten Fall ist eine solche Neuberechnung für den aktuellen Zustand aufgrund eines neu erreichten Folgezustands erforderlich.

Algorithmus 1 zeigt den Pseudocode für die neue Prozedur updateUnsafeZones: Die aufgerufene Prozedur computeUnsafeZones (Z. 4) setzt die Berechnung der unsicheren Zones nach der Definition aus dem vorhergehenden Abschnitt um. Diese Berechnung erfolgt für den als Parameter gegebenen SG-Zustand s abhängig von seinen ausgehenden Transitionen. Falls dessen initiale Zone sich komplett mit unsicheren Zones überschneidet, so wird `false` zurückgegeben, um diesen Spezialfall anzuzeigen (Z. 6). Sind die neu berechneten unsicheren Zones für s unterschiedlich zu den bisherigen (Z. 8), so wird updateUnsafeZones für alle seine direkten Vorgängerzustände (also die Ausgangszustände $t_{in}.source$ der eingehenden Transitionen $s.incoming$) rekursiv aufgerufen (Z. 10). Wenn sich die unsicheren Zones für einen Zustand nicht geändert haben, dann terminiert die Prozedur ohne (weiteren) rekursiven Aufruf. Die Prozedur gibt `false` zurück, sofern irgendeiner der rekursiven Aufrufe `false` zurückgegeben hat.

Algorithm 1 Die Prozedur updateUnsafeZones des Algorithmus für zeitbehaftete verteilte Synthese

```
 1: procedure updateUnsafeZones(s)
 2:     ok ← true
 3:     old_unsafe ← s.unsafe
 4:     s.unsafe ← computeUnsafeZones(s)
 5:     if s.initial - s.unsafe = false then
 6:         return false
 7:     end if
 8:     if old_unsafe ≠ s.unsafe then
 9:         for all t_in ∈ s.incoming do
10:             ok ← ok ∧ updateUnsafeZones(t_in.source)
11:         end for
12:     end if
13:     return ok
14: end procedure
```

Durch Zyklen im SG kann der rekursive Aufruf von updateUnsafeZones bereits betrachtete SG-Zustände erneut erreichen. Dennoch terminiert updateUnsafeZones

schließlich: Ein rekursiver Aufruf für die Vorgängerzustände erfolgt nur bei geänderten unsicheren Zones (Z. 8). Diese können durch updateUnsafeZones jedoch immer nur erweitert werden, wobei die Erweiterung zwangsläufig in ganzzahligen Schritten erfolgt, da Uhren in Clock Zones nur mit ganzzahligen Werten verglichen werden. Daher führt der Aufruf von updateUnsafeZones schließlich dazu, dass alle erreichten Zustände je eine Menge von unsicheren Zones haben, die der Definition von `unsafe(s)` entspricht und die sich für den aktuellen Controllersystem-Kandidaten nicht mehr ändert.

Entfernen von nicht Annahmen-konformem Umgebungsverhalten

Unsichere Zones sind nur in den Zeitbereichen relevant, die in einer den Annahmen entsprechenden Umgebung erreicht werden können. Hier wird erläutert, wie die Zeitbereiche ermittelt werden, für die dies *nicht* der Fall ist.

Analog zu den unsicheren Zones berechnet der Algorithmus diejenigen Clock Zones, in denen die Umgebung eine Verletzung der Umgebungsannahmen nicht mehr vermeiden kann. Diese werden hier als *ausgeschlossene Zones* bezeichnet. Das System muss bei unkontrollierbaren Transitionen nicht damit rechnen, dass die Umgebung diese in einem Zeitraum schaltet, in dem das Schalten zum Erreichen einer unsicheren Zone im Folgezustand führt. Die Synthese braucht daher für dieses Umgebungsverhalten kein Systemverhalten zu erzeugen. Daher werden ausgeschlossenen Zones in SG-Zuständen von den unsicheren Zones desselben Zustands subtrahiert. Es gibt innerhalb der ausgeschlossenen Zones also kein unsicheres Verhalten, da dieses den Annahmen nach ohnehin nicht erreicht werden kann.

Bei Erreichen eines SG-Zustands mit verletzten Annahmen wird für diesen eine ausgeschlossene Zone ohne Einschränkungen (`true`) erzeugt. In Zeitentscheidungszuständen entstehen ausgeschlossene Zones außerdem immer dann, wenn es in diesen ein Zeitentscheidungsereignis gibt, das direkt zu einer Verletzung der Annahmen führt (z. B. durch eine heiße Violation wegen einer verletzten Zeitbedingung in einem Assumption MSD). Dann wird eine ausgeschlossene Zone erzeugt, die der Bedingung dieses Ereignisses entspricht. Entstehen in einem Zustand ausgeschlossene Zones, so werden diese ähnlich wie unsichere Zones zu den Vorgängerzuständen propagiert (siehe vorheriger Abschnitt). Auch die ausgeschlossenen Zones werden basierend auf den ausgehenden Transitionen und abhängig vom Zustandstyp berechnet.

Die Berechnung der ausgeschlossenen Zones für Zeitentscheidungszustände entspricht im Wesentlichen der Berechnung der unsicheren Zones für diese Zustände. Der einzige Unterschied ist, dass für nicht implementierte Ereignisse keine neuen bzw. erweiterten ausgeschlossenen Zones erzeugt werden, sondern für Ereignisse, welche die Annahmen verletzen (s. o.). Aufgrund der Ähnlichkeit zur Berechnung der unsicheren Zones wird hier auf eine explizite Definition für diesen Fall verzichtet. Der Fall ohne Zeitentscheidungsereignisse unterscheidet sich jedoch deutlicher und wird nachfolgend neu definiert.

In einem SG-Zustand *ohne* Zeitentscheidungsereignisse s mit ausgehenden implementierten Transitionen T_{imp} ist die Federation der ausgeschlossenen Zones `excluded(s)`:

$$excluded(s) = \bigwedge_{t \in T_{imp}} \left(\bigvee_{z \in excluded(target(t))} adapt(z, s, target(t)) \right) \tag{4.8}$$

Da die Umgebung bestimmt, welches von mehreren unkontrollierbaren Ereignissen in einem Zustand auftritt, kann sie ausgeschlossene Zones in den Folgezuständen einzelner unkontrollierbarer Transitionen vermeiden. Daher müssen die Folgezustände *aller* dieser Transitionen überlappende ausgeschlossene Zones haben, damit eine ausgeschlossene Zone im Ausgangszustand entsteht. Dasselbe gilt bei mehreren implementierten kontrollierbaren Transitionen, da auch hier die Umgebung die Auswahl trifft. Die ausgeschlossenen Zones haben Auswirkungen darauf, ob unkontrollierbare Transitionen durch das Schalten kontrollierbarer Transitionen verhindert werden können.

Abbildung 4.11 zeigt beispielhaft die Berechnung ausgeschlossener Zones für einen SG-Zustand `s` mit drei implementierten Ereignissen. Für die ausgeschlossenen Zones `c > 2`, `c > 4` und `c < 5` wird hier als Schnittmenge `4 < c < 5` ermittelt.

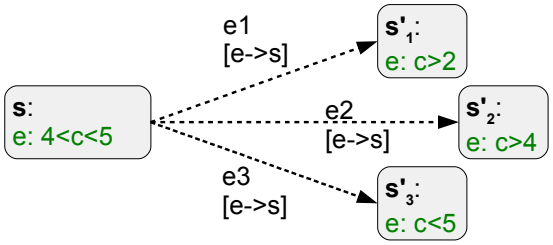


Abbildung 4.11: Beispiel für die Berechnung von ausgeschlossenen Zones für gewöhnliche SG-Zustände

Basierend auf den ausgeschlossenen Zones wird nachfolgend die Verhinderbarkeit von Transitionen (siehe Abschnitt 4.2.4) konkret auf Clock Zones bezogen.

Eine unkontrollierbare SG-Transition t_u ist *verhinderbar* durch einen Controller, wenn für diesen kontrollierbare ausgehende Transitionen T_c im Ausgangszustand s von t_u existieren, sodass gilt:

- t_u kann niemals vor allen kontrollierbar sicheren Zones schalten:
  $included(target(t_u)) - c\_safe(s)^{\uparrow} = false$ und
- t_u kann niemals bereits in der initialen Zone von s schalten, ohne dass gleichzeitig kontrollierbar sichere Zones erreichbar sind:
  $included(target(t_u)) \wedge initial(s) - c\_safe(s)) = false$ und
- s kann nicht *nach* allen kontrollierbar sicheren Zones, aber *vor* t_u betreten werden:
  $\left(initial(s) - c\_safe(s)^{\downarrow}\right)^{\uparrow} \wedge included(target(t_u)) = false.$

Dabei sei `included(s)` $= initial(s)^{\uparrow} - excluded(s)$ die Federation der *nicht* ausgeschlossenen Zones für Zustand s. Die übrigen Funktionen sind in Abschnitt 4.3.3 definiert.

4.3.4 Extraktion der zeitbehafteten Controller

Die im vorherigen Abschnitt beschriebenen Erweiterungen des Synthesealgorithmus bewirken, dass dieser für eine gegebene zeitbehaftete MSD-Spezifikation ein Controllersystem erzeugt, das die Spezifikation erfüllt – sofern diese realisierbar ist. Das erzeugte

Controllersystem selbst enthält jedoch keine Informationen über das zur Einhaltung der Spezifikation erforderliche Echtzeitverhalten. Diese Informationen liegen nach Terminierung des Algorithmus in Form der unsicheren Zones aller erreichbaren Zustände des SG vor. Daraus lassen sich die zeitbehafteten Einschränkungen des sicheren Systemverhaltens ableiten. Um aber für jedes Systemobjekt ein komplettes Implementierungsmodell einschließlich des zeitbehafteten Verhaltens zu erhalten, muss dieses sichere Verhalten den einzelnen Controllern zugeordnet und in Form konkreter zeitbehafteter Modellelemente umgesetzt werden.

Dieser Abschnitt beschreibt zunächst, wie aus den nach Terminierung des Algorithmus vorliegenden unsicheren Zones (bzw. den daraus ableitbaren sicheren Zones) konkrete Modellelemente der Timed Automata in den einzelnen Controllern erzeugt werden können. Dabei werden diese Modellelemente so erzeugt, dass die resultierenden zeitbehafteten Controller nur das durch die Synthese ermittelte erlaubte Echtzeitverhalten umsetzen.

Die so ergänzten zeitbehafteten Controller enthalten oft unnötige Zustände und Transitionen, die eine manuelle Weiterentwicklung erschweren. Daher wird anschließend erläutert, wie diese unnötigen Modellelemente identifiziert und entweder entfernt oder zusammengefasst werden können.

Erzeugung der Clock Resets, Time Guards und Invarianten

Die zeitbehafteten Controller (siehe Abschnitt 2.2.2) müssen so erzeugt werden, dass bei ihrer Ausführung in einer Annahmen-konformen Umgebung niemals unsichere Zones erreichbar sind. Die Synthese stellt bereits sicher, dass dies prinzipiell möglich ist, also dass das System beispielsweise nicht schon initial in einer unsicheren Zone ist und dass die Vermeidung von unsicheren Zones ohne Deadlock möglich ist. Das Sendeverhalten der Controller muss nun so eingeschränkt werden, dass der jeweils aktuelle SG-Zustand vor Erreichen einer unsicheren Zone verlassen wird, und der Folgezustand nur außerhalb einer unsicheren Zone betreten werden kann. Ersteres kann durch Invarianten bewirkt werden, letzteres durch Time Guards (ggf. in Kombination mit Invarianten).

Die durch die Synthese ermittelten unsicheren Zones beziehen sich auf die in den MSDs definierten globalen Uhren. Diese Uhren sind jedoch zunächst nur auf Ebene der MSD-Spezifikation definiert, und existieren normalerweise nicht im erzeugten System. Für verteilte Systeme sind globale Uhren oft schwer umzusetzen, insbesondere wenn sie den Zeitabstand zwischen Ereignissen an verschiedenen Stellen im System modellieren. Daher wird hier davon ausgegangen, dass die einzelnen Controller lokale Uhren besitzen und ihr Echtzeitverhalten in Bezug auf diese definiert wird. Da sich die Grenzen der unsicheren Zones jedoch auf die globalen Uhren beziehen, müssen die Werte der lokalen Uhren denen der globalen entsprechen – zumindest immer dann, wenn ein Time Guard oder eine Invariante das Überschreiten einer solchen Grenze verhindern soll.

Bei der Extraktion der Controller werden die lokalen Uhren der Controller statisch definiert und existieren damit ab dem Systemstart. Auf diese Weise bleiben die Controller kompatibel zum Modell der Timed Automata, in denen zur Laufzeit keine neuen Uhren erzeugt werden können. Zunächst wird für jede globale Uhr in jedem Controller, der diese beobachten kann, eine lokale Uhr erzeugt. Wenn für eine lokale Uhr eines

Controllers jedoch keine Time Guards oder Invarianten erzeugt werden, dann wird diese Uhr wieder entfernt.

Clock Resets Der Wert einer lokalen Uhr eines Controllers entspricht immer dann dem Wert einer globalen Uhr, wenn beide Uhren zuletzt zum gleichen Zeitpunkt zurückgesetzt bzw. erzeugt wurden. Dazu muss ein Clock Reset im lokalen Controller so angelegt werden, dass die lokale Uhr gleichzeitig zum Clock Reset der globalen Uhr zurückgesetzt wird. Der letzte Clock Reset einer globalen Uhr ist immer abhängig von einem Ereignis. Damit ein Controller seine lokale Uhr zum gleichen Zeitpunkt zurücksetzen kann, muss er dieses Ereignis (und damit die Uhr, vgl. Abschnitt 4.2.4) also beobachten können. Er enthält dann eine Transition, die das Ereignis sendet oder empfängt. An dieser Transition wird der Clock Reset angelegt.

Ein Clock Reset einer globalen Uhr kann anhand der Ausgangs- und Folgezustände von SG-Transitionen festgestellt werden: Eine Uhr wird durch eine Transition dann zurückgesetzt, wenn die Uhr in der initialen Zone des Ausgangszustands nicht beschränkt war, aber in der initialen Zone des Folgezustands auf genau 0 beschränkt ist. Lokale Uhren müssen jedoch auch dann zurückgesetzt werden, wenn ihre globale Entsprechung neu instanziiert wird. In einem solchen Fall ist die Uhr im Folgezustand der SG-Transition vorhanden, aber nicht im Ausgangszustand.

Time Guards Time Guards werden für Transitionen angelegt, um zu verhindern, dass durch zu frühes oder zu spätes Senden einer Nachricht eine unsichere Zone im Folgezustand erreicht wird. Durch Negation der unsicheren Zones eines SG-Zustands können die sicheren Zones ermittelt werden, also die Clock Zones, in denen der Zustand betreten werden darf. Diese Berechnung wird für die korrespondierenden Zustände der Folgezustände aller sendenden Controllertransitionen durchgeführt. Eine Controllertransition darf nur schalten, während alle korrespondierenden Zustände des Folgezustands in einer sicheren Zone sind. An jeder Controllertransition wird für den frühest möglichen Schaltzeitraum ein Time Guard angelegt.

Für empfangende Transitionen werden zunächst keine Time Guards angelegt, da der Sender den Sendezeitpunkt festlegt und der Empfänger das Senden nicht blockieren darf (siehe Abschnitt 2.2.2). Durch das spätere Zusammenfassen mit Zeitentscheidungstransitionen (siehe nächster Abschnitt) können jedoch Time Guards von diesen übernommen werden.

Invarianten Invarianten an einem Zustand zwingen einen zeitbehafteten Controller zum Verlassen des Zustands bevor die Invariante ungültig wird. Mittels Invarianten wird verhindert, dass durch Warten im aktuellen SG-Zustand eine unsichere Zone erreicht wird. Dazu wird zunächst die obere Schranke des frühesten sicheren Zeitraums ermittelt (obere Schranke von $(initial(s)^{\uparrow} - unsafe(s)^{\uparrow})^{\downarrow}$ für SG-Zustand s). Diese obere Schranke wird dann als Invariante in allen korrespondierenden Controller-Zuständen übernommen, sofern aus diesen Zuständen sendende Transitionen ausgehen.

Weiterhin werden Invarianten auch in Fällen angelegt, in denen unabhängig von der Erreichbarkeit unsicherer Zones ein Deadlock auftreten könnte: In einem Controller-

zustand ohne Invarianten darf der Controller beliebig lange verweilen. Es wird zwar angenommen, dass der Controller irgendwann sendet (siehe Abschnitt 2.2) – das gilt aber nur, solange der Controller noch senden kann. Wenn die betreffenden Transitionen also Time Guards mit oberen Schranken haben, dann kann der Controller so lange warten, bis keine der Transitionen mehr schaltbar ist. Eine andere Art von Deadlock kann entstehen, wenn zwar eine Invariante existiert, der Controller aber bei Erreichen der durch sie definierten oberen Schranke keine sendenden ausgehenden Transitionen mehr schalten kann. Er verletzt dann zwangsläufig die Invariante, was auch als „Time Stopping Deadlock“ bezeichnet wird [BGS05].

Um diese Deadlocks zu verhindern, werden in jedem Controllerzustand mit ausgehenden sendenden Transitionen Invarianten angelegt, die den oberen Schranken der zuletzt schaltbaren Transitionen dieser Art entsprechen. Invarianten werden auch in Controller-Zuständen angelegt, die Synchronisationsnachrichten senden. Dabei gelten dieselben Regeln wie bei gewöhnlichen Nachrichten.

Behandlung unbeobachtbarer Uhren Ist eine Uhr für einen Controller in einem SG-Zustand nicht beobachtbar, dann kann dieser Controller in den korrespondierenden Zuständen kein zeitabhängiges Verhalten in Bezug auf diese Uhr definieren – also keine Time Guards oder Invarianten. Sofern diese Uhr durch Warten Werte erreichen kann, die das Erreichen einer unsicheren Zone bewirken, kann der Controller also nicht „wissen“, wann er den Zustand verlassen muss. Der Algorithmus identifiziert diese Nachrichten als nicht verzögerbar (vgl. Abschnitt 4.3.2). In diesen Fällen muss der Controller den betreffenden Zustand daher so bald wie möglich über eine ausgehende Transition verlassen, um ein Erreichen der unsicheren Zone zu vermeiden.

Sofern für mindestens eine ausgehende Transition eines solchen Zustands kein Time Guard mit unterer Schranke ($>$, $\geq$ oder $=$) definiert ist, wird der Zustand als *urgent* markiert (siehe Abschnitt 2.2), sodass der Controller diesen direkt nach dem Betreten verlassen muss. Abbildung 4.12 zeigt ein Beispiel, in dem ein Systemobjekt `s1` durch Senden der Nachricht `m1` einen Clock Reset auslöst. Das Systemobjekt `s2` kann diesen jedoch nicht beobachten und „weiß“ daher nach Empfang von `m2` nur, dass `m3` gesendet werden muss, aber nicht wann. Tatsächlich wäre ein Senden bis `c = 3` erlaubt, was aber ohne Kenntnis des Werts der Uhr `c` nicht feststellbar ist. Daher muss `s2` sofort senden, was durch die Markierung von `s2.2` als urgent (mittels „u“ am Zustand) modelliert wird.

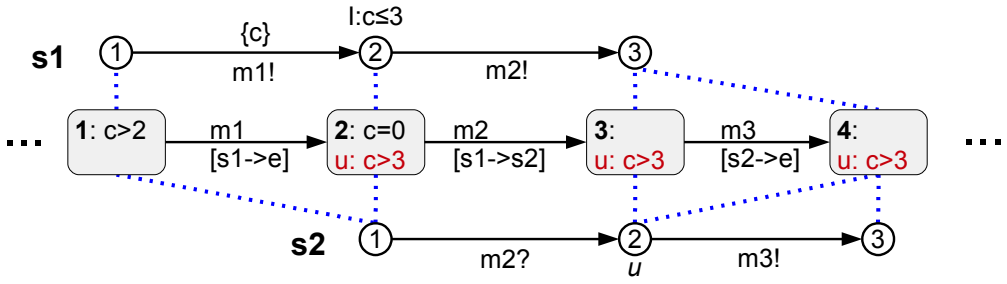


Abbildung 4.12: Beispiel für die Behandlung unbeobachtbarer Uhren

Falls ein Zustand frühestmöglich verlassen werden muss, aber jede ausgehende Transition einen Time Guard mit unterer Schranke definiert, so wird in diesem Zustand eine Invariante angelegt, die ein Schalten erzwingt, sobald einer der Time Guards erfüllt ist. Ist beispielsweise eine Transition mit Time Guard $\mathtt{c} \geq \mathtt{3}$ vorhanden, so wird die Invariante auf $\mathtt{c} \leq \mathtt{3}$ gesetzt.

Das Problem der nicht verzögerbaren Ereignisse könnte alternativ auch durch zusätzliche Synchronisationsnachrichten gelöst werden, durch die ein Controller, der die betreffende globale Uhr beobachten kann, den aktuellen Uhrwert einem Controller mitteilt, der die Uhr nicht beobachten kann. Dadurch würde die Uhr für den Controller, der die Nachricht empfängt, beobachtbar – bis zum nächsten unbeobachtbaren Clock Reset. Dieser Ansatz wäre jedoch komplexer und erforderte mehr Kommunikationsaufwand.

Eine weitere Alternative ist in Fällen möglich, in denen der Wert einer beobachtbaren Uhr immer eine bestimmte Differenz zu dem der nicht beobachtbaren Uhr hat. Die Time Guards und Invarianten könnten dann entsprechend modifiziert für die beobachtbare Uhr angelegt werden. Dieser Ansatz ist jedoch nur in Spezialfällen möglich, in denen eine passende alternative Uhr existiert, die beobachtbar ist.

Zusammenfassen redundanter Modellelemente

Die zeitbehaftete Synthese erzeugt zunächst eigene Controllertransitionen für beobachtbare Verzögerungsereignisse und Zeitentscheidungsereignisse. Nach der Erzeugung der entsprechenden Time Guards und Invarianten sind diese Transitionen und alle Controllerzustände, die nur solche ausgehenden Transitionen haben, oft redundant. Sie werden daher nach der zeitbehafteten Synthese und der Erzeugung der zeitbehafteten Modellelemente in den Controllern aus diesen entfernt bzw. mit anderen Transitionen zusammengefasst. Weiterhin werden auch redundante lokale Uhren zusammengefasst.

Wenn in einem Controllerzustand nur eine ausgehende Transition existiert und diese ein Zeitentscheidungsereignis hat, dann können diese Transition und der Zustand entfernt werden. Die eingehenden Transitionen des entfernten Zustands werden dann auf den Folgezustand der entfernten Transition umgeleitet. Diese Modifikation ändert das Kommunikationsverhalten des Controllers nicht, da die Bedingung des Zeitentscheidungsereignisses bereits nach dem Schalten der eingehenden Transitionen erfüllt sein muss. Dies ergibt sich daraus, dass jedes Zeitentscheidungsereignis ohne Zeitverlust unmittelbar nach einem anderen Ereignis auftritt – also zu demselben Zeitpunkt (vgl. Abschnitt 4.2.3).

Wenn ein Controllerzustand nur eine eingehende Transition hat und diese keine Clock Resets, aber ein Verzögerungsereignis hat und wenn der Zustand eine ausgehende Transition für ein Nachrichtenereignis hat, dann kann dieser Zustand entfernt werden. Die beiden ein- und ausgehenden Transitionen werden dann zu einer einzigen zusammengefasst, die vom Ausgangszustand der eingehenden Transition zum Folgezustand der ausgehenden führt. Das Ereignis ist dasselbe wie bei der ausgehenden Transition. Vorhandene Time Guards der ausgehenden Transition und Invarianten des Zwischenzustands werden übernommen und mit dem Time Guard der Transition für das Verzögerungsereignis konjunktiv verknüpft. Dadurch kann das Nachrichtenereignis in demselben Zeitraum wie zuvor gesendet bzw. empfangen werden.

Gibt es Zustände, bei denen Folgezustand und Ereignis für alle ausgehenden Transitionen identisch sind, dann werden diese Zustände zusammengefasst. Dazu werden alle diese Zustände bis auf einen entfernt. Alle eingehenden Transitionen der entfernten Zustände erhalten als neuen Folgezustand den verbleibenden Zustand. Alle ausgehenden Transitionen der entfernten Zustände werden ebenfalls entfernt. Durch Zusammenfassen von Zuständen kann das Zusammenfassen weiterer Zustände ermöglicht werden. Diese Vereinfachung kann auch für nicht zeitbehaftete Controller durchgeführt werden.

Lokale Uhren können immer dann zu einer einzigen zusammengefasst werden, wenn sie ausschließlich zu denselben Zeitpunkten zurückgesetzt werden, da dann ihr Wert identisch ist. Dabei ist ein Clock Reset nur dann relevant, wenn nach diesem die jeweilige Uhr in einer Invariante oder einem Time Guard verglichen wird, bevor die Uhr erneut zurückgesetzt wird. Beim Zusammenfassen werden die betreffenden Uhren entfernt und eine neue Uhr wird erzeugt, deren Name die Namen der entfernten Uhren enthält. Alle Invarianten, Clock Resets und Time Guards, die sich zuvor auf eine der entfernten Uhren bezogen, beziehen sich nun auf diese neue Uhr. Werden dabei mehrere dieser Elemente identisch, so werden die Duplikate entfernt.

4.3.5 Korrektheit

Im Folgenden wird informell für die Korrektheit des für Echtzeitsysteme erweiterten Algorithmus argumentiert. Grundsätzlich gilt dabei dieselbe Argumentation wie sie in Abschnitt 3.3.5 für den nicht zeitbehafteten Synthesealgorithmus geführt wurde. Aufgrund der in den vorherigen Abschnitten beschriebenen Änderungen und der unterschiedlichen Definition einer Lösung (vgl. Abschnitte 4.3.1 und 3.2.2) wird hier kurz diskutiert, warum diese Argumentation auch für den zeitbehafteten Algorithmus gilt.

Die beschriebenen Änderungen des Algorithmus gefährden die *Terminierung* nicht, da sie keine zusätzlichen Iterationen des Algorithmus verursachen und da die neu eingeführten Operationen terminieren. Die geänderten Kriterien für Implementiertheit (siehe Abschnitt 4.3.1) können zwar bewirken, dass Lösungskandidaten früher verworfen werden; die betreffenden Kandidaten werden jedoch im Kandidatengraph als verworfen vermerkt und werden daher nicht erneut evaluiert. Die Prozedur `updateUnsafeZones` ruft sich selbst rekursiv auf, für ihre Terminierung wurde jedoch bereits in Abschnitt 4.3.3 argumentiert. Die übrigen Berechnungen der unsicheren Zones enthalten keine rekursiven Aufrufe und nur Zyklen, die über eine endliche Menge von Elementen iterieren. Wenn im System die möglichen Differenzen der Uhrenwerte unbeschränkt sind, dann wird der zeitbehaftete SG jedoch unendlich groß. Um Terminierung zu garantieren, muss also angenommen werden, dass die Uhrendifferenzen beschränkt sind.

Auch die *Korrektheit der ermittelten Lösung* ist aus denselben Gründen wie bei der nicht zeitbehafteten Variante gewährleistet: Durch die Anpassung der Definition der Implementiertheit gibt der Algorithmus nur dann einen Lösungskandidaten als Lösung zurück, wenn dieser einer zeitbehafteten Gewinnstrategie entspricht (siehe Abschnitt 4.3.1) und daher von jedem erreichbaren SG-Zustand aus ein Zielzustand erreichbar ist. Zeitbereiche, in denen das nicht gewährleistet ist, werden durch die Berechnung der unsicheren Zones ermittelt. Die geänderten Kriterien für Implementiertheit garantieren zusammen mit der Rückpropagierung der unsicheren Zones (siehe Abschnitt 4.3.3), dass Lösungs-

kandidaten verworfen werden, für die das Erreichen einer unsicheren Zone nicht verhindert werden kann.

Auch die zeitbehaftete Variante des Algorithmus erzeugt systematisch alle infrage kommenden Kandidaten. Die Argumentation für *Vollständigkeit* aus Abschnitt 3.3.5 gilt daher weiterhin.

4.3.6 Beispielausführung

Dieser Abschnitt erläutert die durch den zeitbehafteten Synthesealgorithmus ausgeführten Schritte anhand einer Ausführung auf die in Abschnitt 3.1 eingeführte und in Abschnitt 4.1 um Echtzeitanforderungen erweiterte MSD-Spezifikation für das Anwendungsbeispiel der Produktionszelle. Dabei wird nicht mehr näher auf die Schritte des Beispiels eingegangen, die bereits für das nicht zeitbehaftete Beispiel in Abschnitt 3.3.6 vorgestellt wurden. Stattdessen liegt der Fokus hier auf denjenigen Schritten, die durch die Echtzeit-Anpassungen verändert werden oder hinzukommen.

Im Folgenden wird die Ausführung des zeitbehafteten Synthesealgorithmus für das Beispiel bis zur Feststellung gezeigt, dass der aktuelle Kandidat eine Lösung ist. Anschließend wird beschrieben, wie auf dieser Basis die endgültigen zeitbehafteten Controller extrahiert werden.

Ausführung des angepassten Algorithmus

In den Abbildungen 4.13 und 4.14 sind einige Zwischenstände (bzw. Ausschnitte daraus) des erweiterten Algorithmus dargestellt. Für eine Erläuterung des Aufbaus der Abbildung und der Bedeutung der Bildelemente ohne Echtzeitbezug wird auf Abschnitt 3.3.6 verwiesen. Wie bisher in diesem Kapitel gibt der Ausdruck nach der Zustandsnummer jedes SG-Zustands jeweils die initiale Zone des Zustands an. Dabei werden hier die Einschränkungen der Uhrendifferenzen in allen Clock Zones aus Platzgründen ausgelassen (vgl. Abschnitt 4.2.2). Die unsicheren Zones werden durch ein vorangestelltes „u:“, die ausgeschlossenen Zones durch ein „e:“ gekennzeichnet.

Zwischenstand a: Wie im nicht zeitbehafteten Fall werden zunächst beide unkontrollierbaren Transitionen in SG-Zustand S.0 implementiert. Im Fall des Ereignisses `intactBlank` wird im Folgezustand S.2 das MSD für die Anforderung **R7** (siehe Abschnitt 4.1) mit der Uhr cIB aktiv, die somit in der initialen Zone von S.2 den Wert 0 hat. Sie modelliert den Zeitabstand seit der letzten Nachricht `intactBlank`.

Auch hier werden die Zustände S.1 und S.2 durch sendende Transitionen für `removeBlank` bzw. `blankToPress` im Controller für aC implementiert. Im Fall von S.1 wird dabei der Zyklus zu S.0 geschlossen.

Als Folgezustand von S.2 wird S.3 erreicht, in dem die Nachricht `blankToPress` das MSD zu **R6** aktiviert. Das MSD fordert eine Wartezeit von mindestens 3 Sekunden vor der nächsten Nachricht. Wie bereits in Abschnitt 4.2.2 diskutiert wurde, wird diese Wartezeit durch das Verzögerungsereignis `d: cBtP` $\geq$ 3 repräsentiert. Da die Nachricht `blankToPress` an die Umgebung gesendet wurde, kann zwar aC, aber nicht pC die neue

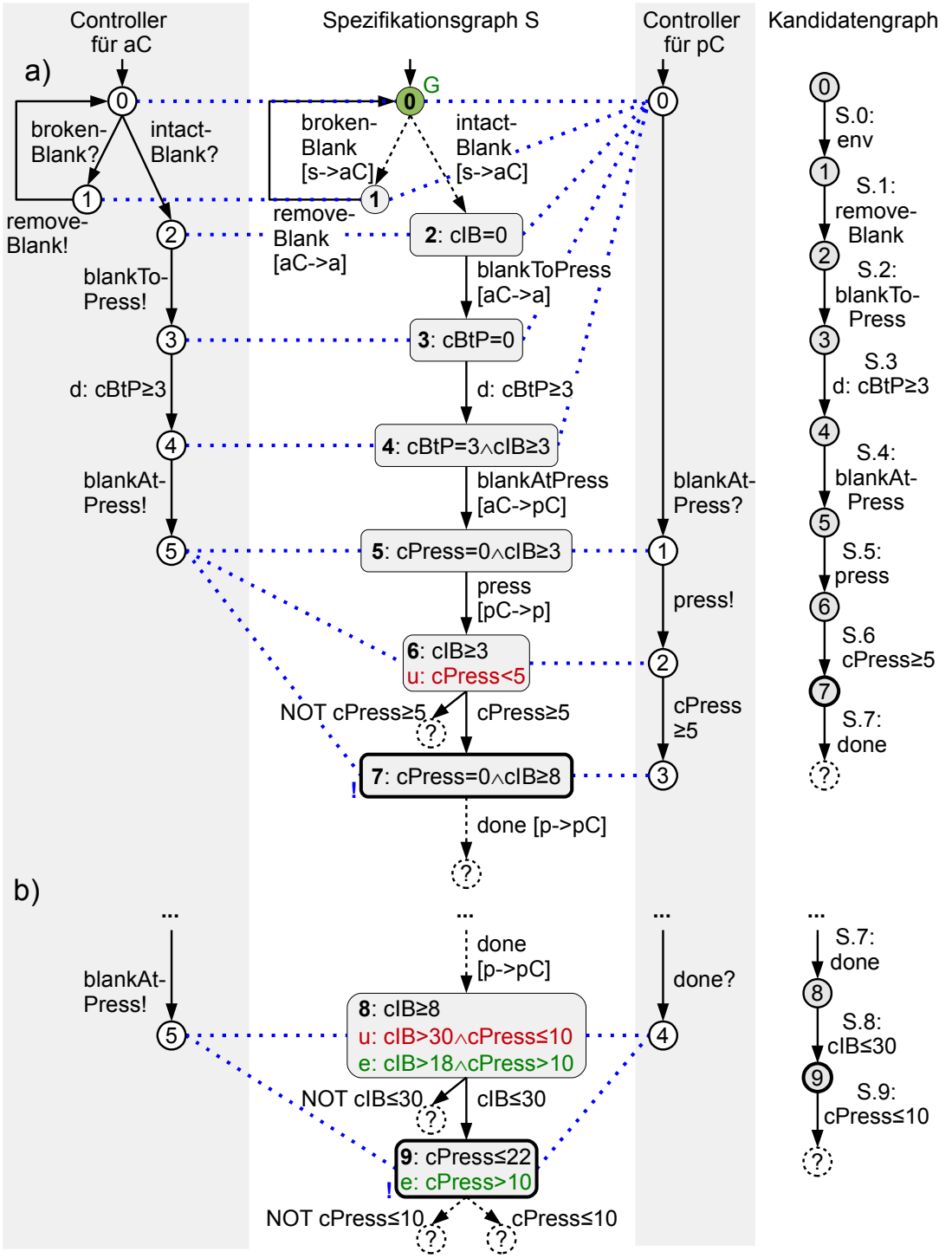


Abbildung 4.13: Beispieldurchlauf für die zeitbehaftete Produktionszelle (Teil 1)

Uhr cBtP beobachten – und damit das Ereignis d: cBtP $\geq$ 3. Das Ereignis wird daher durch eine entsprechende Transition in aC implementiert, an der dieses annotiert wird.

Durch das Warten in S.3 wird S.4 erreicht. Das Ereignis blankAtPress wird, wie im nicht zeitbehafteten Fall, durch sendende bzw. empfangende Transitionen in beiden Controllern implementiert.

Im Folgezustand S.5 wird das MSD zur Annahme **A3** aktiv, die nur zum Tragen kommt, wenn press erst frühestens 5 Sekunden nach blankAtPress gesendet wird. Zunächst wird S.5 aber auch im zeitbehafteten Fall durch eine ausgehende Transition in pC implementiert, die press sendet.

Im Zeitentscheidungszustand S.6 wird über die kalte Zeitbedingung cPress $\geq$ 5 entschieden. Das System bestimmt den Sendezeitpunkt von press und damit den Zeitpunkt des Betretens von S.6. Daher sind die beiden Zeitentscheidungsereignisse kontrollierbar und es braucht nur eines von diesen implementiert werden. Hier wird (nach nichtdeterministischer Auswahl des Algorithmus aus den beiden Zeitentscheidungsereignissen) cPress $\geq$ 5 implementiert.

Der letzte Clock Reset von cPress durch blankAtPress ist für aC und pC beobachtbar. Da die Nachricht press von pC an die Umgebung gesendet wird, kennt jedoch nur pC den Zeitpunkt des Betretens von S.6 und kann daher cPress $\geq$ 5 (und NOT cPress $\geq$ 5) beobachten. Daher wird nur für diesen Controller eine neue Transition angelegt.

Durch Implementieren von cPress $\geq$ 5 entsteht die unsichere Zone cPress < 5 in S.6 für den nicht implementierten Fall. Der Zustand darf also nicht mit Uhrenwerten in dieser Zone betreten werden – also nach weniger als 5 Sekunden Wartezeit zwischen blankAtPress und press.

Der hier nicht betrachtete Fall, dass zunächst NOT cPress $\geq$ 5 implementiert wird, führt nach einigen zusätzlichen Schritten zu demselben Ergebnis: Das MSD zu **A3** wird durch eine kalte Violation beendet, wodurch die Umgebung im Folgezustand beliebig lange vor dem Senden von done warten darf. Dadurch wird die heiße Zeitbedingung cIB $\leq$ 30 in Anforderung **R7** unerfüllbar und ein Backtracking zu S.6 ist erforderlich. Anschließend verbleibt das Implementieren von cPress $\geq$ 5 als einzige Alternative.

Im neu erreichten Zustand S.7 ist die Uhr cPress durch den Clock Reset im MSD zu **A3** wieder 0. Das MSD ist weiter aktiv und schränkt die mögliche Wartezeit der Umgebung in S.7 vor dem Senden von done ein.

Zwischenstand b: Das Ereignis done in Zustand S.7 wird durch eine empfangende Transition im Controller für pC implementiert. Zustand S.8 ist erneut ein Zeitentscheidungszustand, diesmal für die heiße Zeitbedingung cIB $\leq$ 30 in Anforderung **R7**. Da der Fall einer Verletzung der Zeitbedingung cIB $\leq$ 30 zwangsläufig eine Verletzung der Anforderungen darstellt, implementiert die Synthese den Fall der erfüllten Bedingung cIB $\leq$ 30 und eine unsichere Zone cIB > 30 (in der Abbildung durch das spätere Erreichen von S.9 modifiziert) in S.8 entsteht. Die Propagierung der unsicheren Zone zu den Vorgängerzuständen wird nachfolgend anhand von Zwischenstand **c** erläutert.

Da nur aC den letzten Clock Reset von cIB durch intactBlank beobachten konnte, aber nur pC den Zeitpunkt des Betretens von S.8 durch done kennt, kann keiner der Controller die beiden Zeitentscheidungsereignisse cIB $\leq$ 30 und NOT cIB $\leq$ 30 beob-

achten. Die Implementierung von S.8 besteht deshalb lediglich in einer Berechnung des Folgezustands S.9 und einer entsprechenden Anpassung der Zustandskorrespondenzen. Es werden also keine neuen Controllertransitionen erzeugt.

Auch der erreichte Zustand S.9 ist ein Zeitentscheidungszustand. Hier entspricht jedoch das Zeitentscheidungsereignis `NOT cPress` $\leq$ `10` einer Verletzung der Annahme **A3** durch eine heiße Verletzung der entsprechenden Zeitbedingung und braucht daher nicht betrachtet bzw. implementiert zu werden. Zusätzlich entsteht in S.9 eine ausgeschlossene Zone `cPress > 10`. Diese wird zu S.8 propagiert (wird dort zu `cIB > 18` $\wedge$ `cPress > 10`) und schränkt entsprechend die unsichere Zone `cIB > 30` zu `cIB > 30` $\wedge$ `cPress` $\leq$ `10` ein.

Den alternativen Fall `NOT cIB` $\leq$ `30` der verletzten Bedingung in S.8 würde die Synthese nur dann betrachten, wenn sich später herausstellt, dass für `cIB` $\leq$ `30` keine Implementierung möglich ist. Das ist hier jedoch nicht der Fall.

Endergebnis: Abbildung 4.14 **c** zeigt den Zustand der Synthese zum Zeitpunkt der Terminierung. Es gibt also keine unimplementierten Zustände mehr.

Anders als in Zwischenstand **b** in Abbildung 4.13 sind hier für alle Zustände die durch die Implementierung von `cIB` $\leq$ `30` erzeugten unsicheren und ausgeschlossenen Zones eingezeichnet. Diese sind seit dem Erreichen von Zustand S.9 unverändert geblieben. Die Erzeugung der Zones in den Zuständen S.8 und S.9 wurde bereits diskutiert. Im Folgenden wird ihre Weiterpropagierung zu den Vorgängerzuständen beschrieben.

Da der Sendezeitpunkt von `done` in S.7 identisch zum Zeitpunkt des Betretens von S.8 ist, muss die ausgeschlossene Zone aus S.8 in S.7 unverändert übernommen werden: Ein späteres Senden von `done` ist immer eine Verletzung der Annahmen. Die Umgebung kann jedoch zuvor bis zu 10 Sekunden (von `cPress = 0` bis maximal `cPress = 10`) warten, womit auch cIB um den gleichen Wert steigen kann. Daher ist in S.7 bereits ein Wert von `cIB > 20` unsicher, sofern dadurch die Annahmen (`cPress` $\leq$ `10`) nicht verletzt werden. Die gemäß Abschnitt 4.3.3 berechnete unsichere Zone für S.7 modelliert dies.

Da Zustand S.6 ein Zeitentscheidungszustand ist, ist in diesem kein Warten vor dem Betreten von S.7 möglich. Daher werden die Zones aus S.7 in S.6 unverändert übernommen – jedoch nur soweit sie sich nicht auf die Uhr cPress beziehen, die in S.6 nicht existiert (vgl. Beschreibung von `adapt` in Abschnitt 4.3.3): Da die ausgeschlossene Zone eine untere Grenze für die in S.7 zurückgesetzte Uhr cPress definiert, entfällt diese Clock Zone. Die aus S.7 übernommene unsichere Zone wird durch Wegfallen von cPress zu `cIB > 20`. Wie bereits für Zwischenstand **a** erläutert wurde, entsteht zusätzlich durch Implementieren von `cPress` $\geq$ `5` eine unsichere Zone `cPress < 5` in S.6.

Um ein Betreten von S.6 innerhalb der unsicheren Zone `cPress < 5` zu verhindern, muss pC in S.5 vor dem Senden von `press` mindestens 5 Sekunden warten. Da das Ereignis kontrollierbar ist, kann das System den Sendezeitpunkt bestimmen. Die unsichere Zone `cPress < 5` aus S.6 entfällt daher in S.5. Die andere unsichere Zone `cPress > 20` muss jedoch für S.5 übernommen werden, da ein späteres Senden von `press` ein Betreten von S.6 innerhalb derselben unsicheren Zone bedeuten würde. Durch das zur Vermeidung von `cPress < 5` in S.6 erforderliche Warten in S.5 entsteht zudem die unsichere

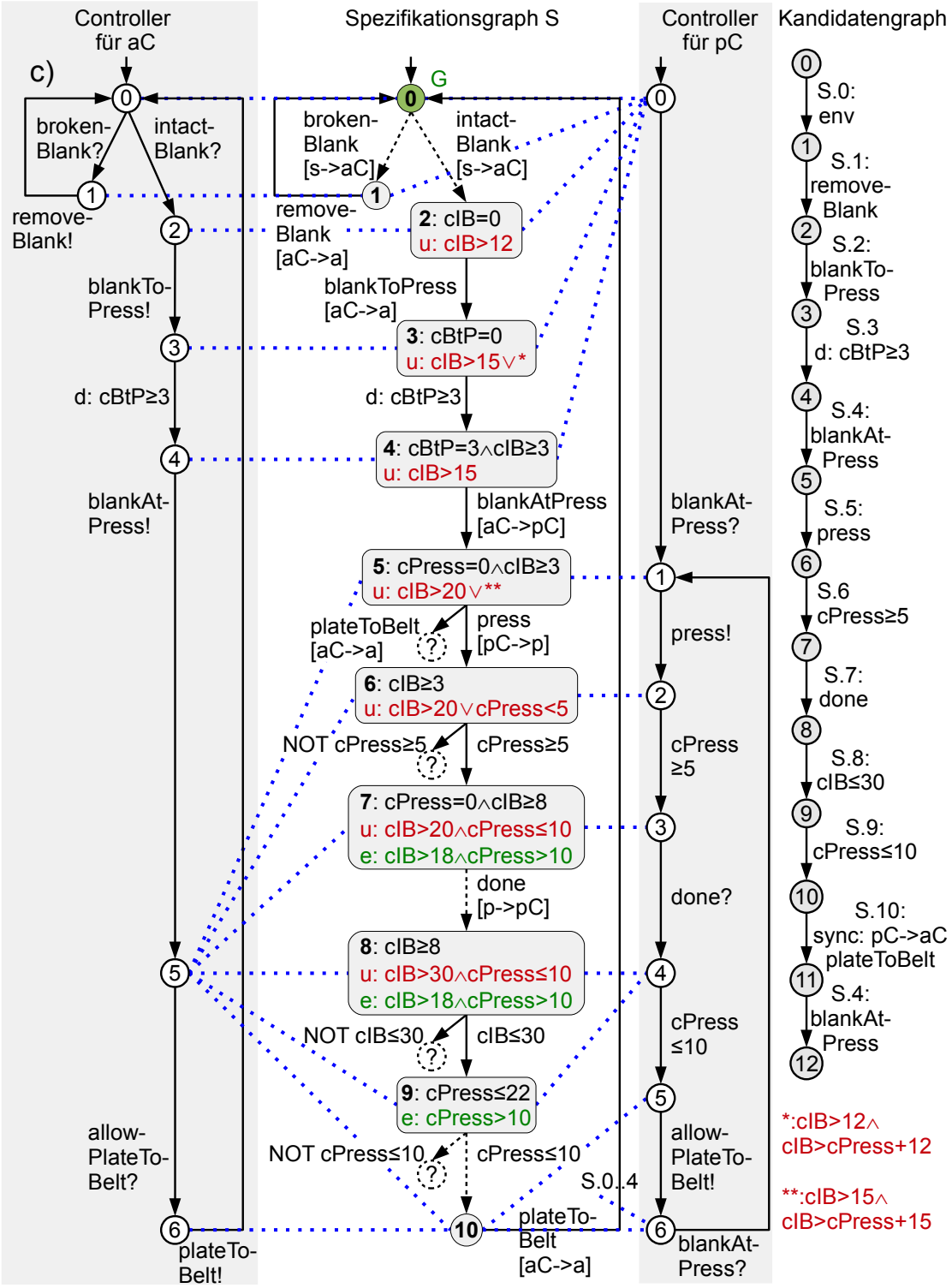


Abbildung 4.14: Beispieldurchlauf für die zeitbehaftete Produktionszelle (Teil 2)

Zone cPress > 15 ∧ cIB > cPress+15 in S.5. Diese modelliert, dass S.5 unsicher ist, wenn der Wert für cIB bereits bei Betreten von S.5 (also bei cPress = 0) so hoch ist, dass kein Warten von 5 Sekunden mehr möglich ist (also bei $cIB > 15$; da $cIB \geq cPress$ gilt, ergibt sich $cPress > 15$).

Um ein Betreten von S.5 innerhalb der unsicheren Zones zu vermeiden, muss in S.4 blankAtPress vor cIB > 15 gesendet werden. Nur so kann in S.5 lange genug gewartet werden, ohne cIB > 20 zu erreichen. Die unsichere Zone in S.4 wurde entsprechend als cIB > 15 berechnet. Dies ergibt sich daraus, dass der Folgezustand zwangsläufig mit cPress = 0 erreicht wird, wodurch die zusätzliche Einschränkung für cIB > 15 entfällt (vgl. Beschreibung von adapt in Abschnitt 4.3.3).

Auch in Zustand S.3 darf das System verweilen, solange cIB > 15 nicht erreicht wird. Zudem muss in diesem Zustand, also seit dem Senden von blankToPress, jedoch mindestens 3 Sekunden gewartet werden. Der Zustand muss daher betreten werden, bevor das Warten zu einem Erreichen von cIB > 15 führt. Daher wird in S.3 zusätzlich die unsichere Zone cIB > 12 ∧ cIB > cBtP+12 erzeugt, die dies modelliert (analog zu S.5).

In Zustand S.2 kann das System ein zwangsläufiges Erreichen der unsicheren Zones in S.3 nur durch rechtzeitiges Senden von blankToPress verhindern. Um die in S.3 erforderliche Wartezeit zu ermöglichen, muss also bereits vor cIB > 12 gesendet werden. Dies entspricht der berechneten unsicheren Zone in S.2 (analog zu S.4).

In Zustand S.0 entsteht keine unsichere Zone, da die Uhren, auf die sich die unsicheren Zones in S.2 beziehen, dort noch nicht existieren. Die Propagierung der unsicheren Zones endet daher.

Nach der Berechnung der ausgeschlossenen und unsicheren Zones nach der Implementierung von cIB ≤ 30 muss der dadurch erreichte Zustand S.9 implementiert werden. Da NOT cPress ≤ 10 den Annahmen widerspricht, muss bzw. kann nur cPress ≤ 10 implementiert werden. Anders als aC kennt pC den Zeitpunkt des letzten Clock Resets von cPress (durch press) und den Zeitpunkt des Betretens von S.9 (der Zeitpunkt von done, da im Zeitentscheidungszustand S.8 keine Zeit vergeht). Da pC cPress ≤ 10 also beobachten kann, wird im Controller für pC eine entsprechende Transition eingefügt.

Zustand S.10 wird wie im nicht zeitbehafteten Fall implementiert und schließt den Zyklus zu S.0. Auch im zeitbehafteten Fall muss blankAtPress durch eine zusätzliche Transition im Controller von pC erneut implementiert werden – diesmal in S.4.

Controller-Extraktion

Abbildung 4.15 illustriert, wie basierend auf dem in Abbildung 4.14 dargestellten Ergebnis der zeitbehafteten Synthese die zeitbehafteten Controller abgeleitet werden. Wie in Abschnitt 4.3.4 beschrieben wurde, werden die zeitbehafteten Modellelemente aus den initialen und unsicheren Zones der zu den Controllerzuständen korrespondierenden SG-Zustände abgeleitet. Diese Modellelemente sind in Abbildung 4.15 hervorgehoben. Anschließend werden überflüssige Zustände und Transitionen entfernt. In Abbildung 4.15 sind die betroffenen Transitionen und Zustände rot markiert und die Namen der entfernten Zeitereignisse durchgestrichen.

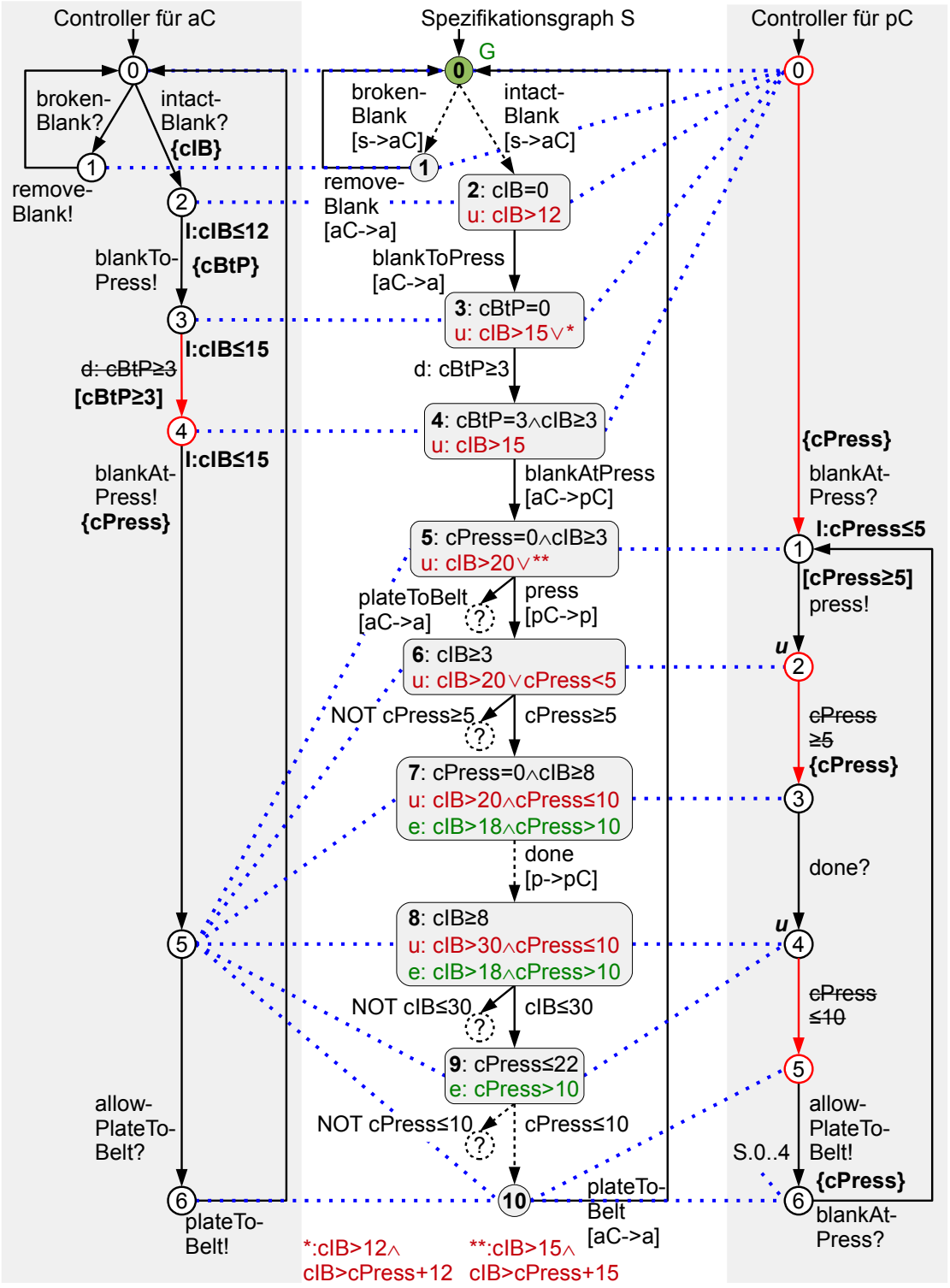


Abbildung 4.15: Controller-Extraktion für die zeitbehaftete Produktionszelle

Anlegen der Clock Resets, Time Guards und Invarianten: Wie in Abschnitt 4.3.4 beschrieben wurde, wird für jede globale Uhr in jedem Controller, der diese beobachten kann, eine entsprechende lokale Uhr angelegt. In diesem Fall sind das cIB, cBtP und cPress für aC und nur cPress für pC. Anhand der initialen Zones der SG-Zustände wird für cIB, cBtP und cPress ermittelt, an welchen Transitionen diese zurückgesetzt werden: In den initialen Zones der Folgezustände dieser Transitionen ist die jeweilige Uhr 0. Hier ist das für cIB in S.2 der Fall, für cBtP in S.3 und für cPress in S.5 und S.7. An den eingehenden Transitionen der zu diesen SG-Zuständen korrespondierenden Controllerzustände in beiden Controllern werden daher entsprechende Clock Resets der lokalen Uhren eingefügt. Dies gilt jedoch nicht, wenn der betreffende Clock Reset für den Controller unbeobachtbar ist, wie der Reset von cPress in S.7 durch die für aC nicht beobachtbare Nachricht `press` – dem letzten Nachrichtenereignis vor dem Clock Reset.

Um die unsichere Zone `cIB > 12` in S.2 zu vermeiden, wird in aC.2 eine Invariante `cIB` $\leq$ `12` angelegt. Analog wird zur Vermeidung von `cIB > 15` in S.3 und S.4 jeweils eine Invariante `cIB` $\leq$ `15` in aC.3 und aC.4 erzeugt.

Um zu frühes Senden von `blankAtPress` zu verhindern, wird für die ausgehende Transition in Zustand aC.3 ein Time Guard `cBtP` $\geq$ `3` erzeugt, der dem Verzögerungsereignis in S.3 entspricht. Analog wird für die ausgehende Transition in pC.1 ein Time Guard `cPress` $\geq$ `5` erzeugt, um ein zu frühes Erreichen von S.6 mit `cPress < 5` zu verhindern.

Da die unteren Schranken der unsicheren Zones in S.5 bis S.8 sich jeweils auf die für pC unbeobachtbare Uhr cIB beziehen, muss pC in den betreffenden Zuständen so schnell wie möglich (in S.5 unter Berücksichtigung der unsicheren Zone `cPress < 5` im Folgezustand) senden. Dies wird für die Zustände pC.2 und pC.4 sichergestellt, indem sie als *urgent* (siehe Abschnitt 2.2.2) gekennzeichnet werden. Der Controller muss diese also so schnell wie möglich verlassen. Zustand pC.1 darf erst nach Ablauf der minimalen Wartezeit verlassen werden. Daher wird hier ein schnellstmögliches Verlassen des Zustands bei Erfüllen des Time Guards durch eine passende Invariante `I: cPress` $\leq$ `5` erzwungen. Ein Verlassen von pC.3 muss bzw. darf nicht erzwungen werden, da der Controller hier auf eine Umgebungsnachricht wartet.

Zusammenfassen redundanter Modellelemente: Alle in Abbildung 4.15 rot markierten Transitionen und Zustände entfallen gemäß der Kriterien in Abschnitt 4.3.4 bzw. sie werden mit anderen Transitionen zusammengefasst. Die ausgehende Transition für pC.4 entfällt beispielsweise, da der Controller nicht auf dieses Zeitentscheidungsereignis reagieren muss. Die ursprüngliche ausgehende Transition des entfernten Zwischenzustands pC.5 führt nun von pC.4 zu pC.6. Die Zwischenzustände aC.4 und pC.2 entfallen ebenfalls und ihre ein- und ausgehenden Transitionen werden jeweils zusammengefasst.

Zusätzlich wird der Startzustand pC.0 mit pC.6 zusammengefasst, da Folgezustand und Ereignis der einzigen ausgehenden Transition in beiden Zuständen identisch sind. Der Zustand pC.0 wird dazu gelöscht und der neue Startzustand ist pC.6.

Abbildung 4.16 zeigt die endgültigen zeitbehafteten Controller nach dem Entfernen der überflüssigen Elemente. Die Nummerierung der Zustände aus Abbildung 4.15 wurde hier für eine bessere Vergleichbarkeit beibehalten.

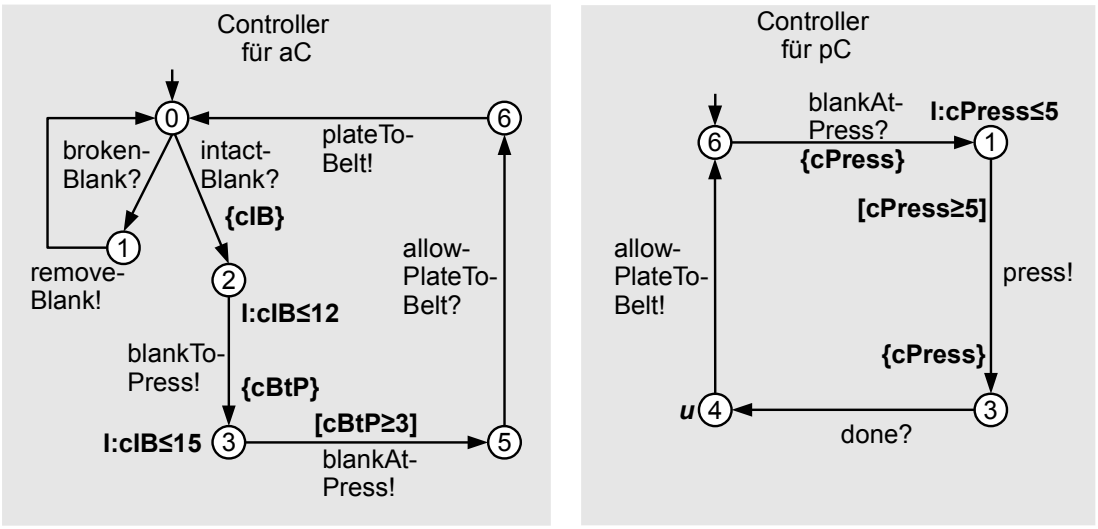


Abbildung 4.16: Durch die Synthese erzeugte zeitbehaftete Controller für die zeitbehaftete Produktionszelle

4.4 Erweiterung für asynchrone Kommunikation mit Zeitbedarf

Die Echtzeit-Erweiterung des Synthesealgorithmus in Abschnitt 4.3 berücksichtigt zwar zeitbehaftete MSD-Spezifikationen, der Zeitbedarf für den Austausch von Nachrichten wird jedoch weiterhin als vernachlässigbar angenommen. In Spezialfällen kann dies aufgrund vernachlässigbarer Übertragungszeiten gerechtfertigt sein, im Allgemeinen kann aber die Dauer der Kommunikation in realen Systemen durchaus Auswirkungen auf die Einhaltung der Anforderungen, insbesondere (aber nicht nur) der Echtzeitanforderungen, haben.

Wenn Nachrichten nicht instantan übermittelt werden, dann ist es aber auch nicht mehr sinnvoll, das Senden und das Empfangen einer Nachricht als ein einziges Ereignis zu modellieren. Insbesondere in verteilten Systemen können Nachrichten mit jeweils unterschiedlichem Zeitbedarf gleichzeitig über verschiedene Kanäle übermittelt werden. Statt der bisher angenommenen synchronen Kommunikation ist also *asynchrone Kommunikation* für verteilte Systeme oft ein angemesseneres Modell. Dieser Abschnitt stellt daher eine Erweiterung des zeitbehafteten Syntheseansatzes für die Berücksichtigung asynchroner Kommunikation mit Zeitbedarf vor.

Grundsätzlich ist es bereits mit dem in Abschnitt 4.3 vorgestellten Algorithmus möglich, asynchrone Kommunikation mit Zeitbedarf zu berücksichtigen. Dafür muss diese jedoch explizit auf Ebene der MSD-Spezifikation modelliert werden, beispielsweise so: Soll die Kommunikation zwischen zwei Objekten asynchron erfolgen, so kann das Kommunikationsmedium zwischen diesen durch ein drittes „Zwischenobjekt“ modelliert werden, das durch die Umgebung gesteuert wird. Jede Kommunikation muss dann über dieses Zwischenobjekt geleitet werden, was durch entsprechende Modifikation der MSDs (Austausch der Empfänger der betroffenen Nachrichten durch das Zwischenobjekt) erreicht werden kann. In zusätzlichen Assumption MSDs kann das Zwischenobjekt dann

gezwungen werden, diese Nachrichten zeitverzögert in einem bestimmten Zeitintervall weiterzuleiten. In Übertragung befindliche Nachrichten werden in Zuständen des Spezifikationsgraphen (SG) dann durch aktive MSDs repräsentiert, in denen eine Nachricht durch das Zwischenobjekt bereits empfangen, jedoch noch nicht weitergeleitet wurde.

Diese Art der Modellierung ist jedoch manuell sehr umständlich. Eine Automatisierung würde neben einer Anpassung der MSD-Spezifikation auch eine Anpassung der erzeugten Controller erfordern. Zudem kann ein Zeitbedarf für zusätzliche Synchronisationsnachrichten so nicht modelliert werden, da sich die MSD-Spezifikation auf diese nicht beziehen kann. Daher löst diese Arbeit das Problem stattdessen durch direkte Anpassungen des Synthesealgorithmus bzw. des zeitbehafteten SG, die jedoch im Vergleich zu den bisher diskutierten zeitbehafteten Erweiterungen relativ geringe Auswirkungen auf den Synthesealgorithmus selbst haben.

Um modellieren zu können, dass die Kommunikation zwischen den Objekten unterschiedlich lange dauern kann und mehrere Nachrichten gleichzeitig übertragen werden können, muss der Modellierungsansatz für Struktur in MSD-Spezifikationen geringfügig erweitert werden. Um die Laufzeitzustände einer so erweiterten Spezifikation geeignet repräsentieren zu können, muss auch der SG bzw. dessen Berechnung erneut angepasst werden. Dies wiederum bedingt auch Anpassungen des Synthesealgorithmus selbst, um separate Sende- und Empfangsereignisse für Nachrichten korrekt behandeln zu können. Weitere Änderungen des Algorithmus sind erforderlich, um den Zeitbedarf von zusätzlich eingefügten Synchronisationsnachrichten zu berücksichtigen.

Abschnitt 4.4.1 stellt zunächst kurz die Erweiterung der Spezifikationstechnik zur Strukturmodellierung vor, die anschließend in Abschnitt 4.4.2 dazu verwendet wird, die Echtzeitvariante des Anwendungsbeispiels erneut zu erweitern. Die Erweiterungen des SG werden in Abschnitt 4.4.3 diskutiert, die entsprechenden Änderungen am Algorithmus in Abschnitt 4.4.4. Schließlich beschreibt Abschnitt 4.4.5 die Anwendung des so erweiterten Algorithmus auf die Beispielspezifikation.

4.4.1 Spezifikation von asynchroner Kommunikation mit Zeitbedarf

Für asynchrone Kommunikation sind einige Eigenschaften der Kommunikationsverbindungen des Systems relevant, die mit den in Abschnitt 2.1.1 eingeführten Mitteln nicht modelliert werden können. Daher werden in diesem Abschnitt die Konnektoren, die diese Kommunikationsverbindungen repräsentieren, so erweitert, dass sie eine Spezifikation dieser Eigenschaften unterstützen.

Für jede Kommunikationsverbindung und für beide Kommunikationsrichtungen kann nun durch eine Beschriftung am entsprechenden Konnektor definiert werden, wie viele Nachrichten sich zwischen den beteiligten Systemen in die jeweilige Richtung in Übertragung befinden können. Diese Zahl wird hier als *Kapazität* bezeichnet. Die in Übertragung befindlichen Nachrichten können sich im realen System beispielsweise in einem Sende- oder Empfangspuffer oder im Kommunikationsmedium befinden.

Wird keine Kapazität bzw. eine Kapazität von 0 definiert, so erfolgt die Kommunikation über die betreffende Verbindung weiterhin synchron. Dadurch ist neben asynchroner Kommunikation weiterhin synchrone Kommunikation möglich und bestehende MSD-Spezifikationen behalten mit der hier beschriebenen Erweiterung dieselbe Semantik.

Die Festlegung einer maximalen Zahl gleichzeitig in Übertragung befindlicher Nachrichten verhindert, dass durch asynchrone Kommunikation ein unendlich großer Zustandsraum entsteht. Bereits relativ kleine Werte können jedoch schon zu einem sehr großen Zustandsraum führen, da jede Kombination in Übertragung befindlicher Nachrichten als eigener SG-Zustand betrachtet wird und die Synthese grundsätzlich alle möglichen Verzahnungen von Sende- und Empfangsereignissen betrachten muss.

Neben der Maximalzahl gleichzeitig übertragener Nachrichten kann für jede Kommunikationsverbindung in jede Richtung ebenfalls per Beschriftung am Konnektor festgelegt werden, wie lange die Übertragung von Nachrichten minimal und maximal dauern kann. Sowohl die maximale Nachrichtenanzahl als auch die Übertragungszeit gelten jeweils für System- und Umgebungsnachrichten. Insbesondere gelten diese Festlegungen auch für zusätzlich zum spezifizierten Kommunikationsverhalten ausgetauschte Synchronisationsnachrichten.

Die Umgebung kann in jedem Fall bestimmen, wie lange die Übertragung jeder konkreten Nachricht innerhalb des definierten Zeitbereichs tatsächlich dauert. Die Synthese muss daher jeweils auch bei ungünstigsten Übertragungszeiten eine Einhaltung der MSD-Spezifikation garantieren. Die an den Konnektoren definierten Eigenschaften der Kommunikationsverbindungen können insgesamt als spezielle Art von Umgebungsannahmen betrachtet werden, die die Eigenschaften der verwendeten Kommunikationskanäle betreffen: Übertragungszeiten außerhalb der festgelegten Intervalle werden als unmöglich angenommen. Ebenso wird angenommen, dass die Umgebung nie Nachrichten sendet, die die für einen Konnektor definierte Kapazität überschreiten würden.

4.4.2 Erweitertes Beispiel

Die in Abschnitt 3.1 eingeführte und in Abschnitt 4.1 erweiterte MSD-Spezifikation für die Produktionszelle wird hier erneut erweitert. Dabei wird die Strukturdefinition aus Abschnitt 3.1.1 um die im vorherigen Abschnitt diskutierten Aspekte ergänzt.

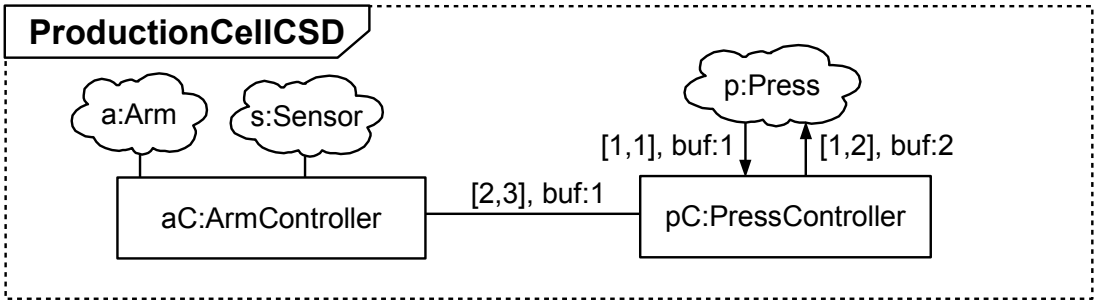


Abbildung 4.17: Struktur der Produktionszelle mit asynchroner Kommunikation

Abbildung 4.17 zeigt die veränderte Struktur des ansonsten unveränderten Beispiels: Für die Kommunikation zwischen einerseits der Presse und ihrem Steuergerät und andererseits zwischen den beiden Steuergeräten wird jeweils ein Nachrichtenpuffer definiert. Über diese Kommunikationskanäle erfolgt die Nachrichtenübertragung nun asynchron,

während die übrige Kommunikation im System weiterhin synchron ist. Für die Kommunikation von der Presse zu ihrem Steuergerät und die Kommunikation zwischen den Steuergeräten wird eine Puffergröße von genau einer Nachricht definiert, während der Puffer für Nachrichten zur Presse eine Kapazität von zwei Nachrichten hat. Der Zeitbedarf der Nachrichten wird variabel definiert: Die Kommunikation zwischen den Steuergeräten aC und pC dauert in beide Richtungen 2 bis 3 Sekunden. Nachrichten vom Steuergerät der Presse pC zur Presse p selbst dauern 1 bis 2 Sekunden. Nachrichten der Presse an ihr Steuergerät kommen immer um exakt eine Sekunde verzögert an.

4.4.3 Erweiterung des zeitbehafteten Spezifikationsgraphen

Senden und Empfangen müssen zur Unterstützung asynchroner Kommunikation als separate Ereignisse behandelt werden. Daher wird der bereits in Abschnitt 4.2 für Echtzeitsysteme erweiterte SG hier um zwei neue Arten von Ereignissen erweitert, die bei der Kommunikation über Konnektoren mit einer Kapazität von mehr als 0 zum Einsatz kommen. Zudem ändert sich die Berechnung der initialen Zones geringfügig.

Da das spätere Systemverhalten von den asynchronen Nachrichten abhängen kann, die sich in Übertragung befinden, müssen diese in den betreffenden SG-Zuständen repräsentiert werden. Dazu wird für jede Kommunikationsverbindung, für die im jeweiligen SG-Zustand Nachrichten im Transit sind, eine eigene Queue angelegt, die diese Nachrichten enthält. In dieser Arbeit wird dabei generell von einer FIFO-Queue (für engl. *First In, First Out*) ausgegangen. Es wird also angenommen, dass alle Nachrichten, die über dieselbe Kommunikationsverbindung gesendet werden, in der Reihenfolge ankommen, in der sie gesendet wurden. Dadurch kann in jedem SG-Zustand immer nur die erste Nachricht jeder Queue empfangen werden.

Bei Auftreten der bereits diskutierten Arten von Ereignissen (Zeitereignisse und Nachrichtenereignisse) behalten die Queues ihren Inhalt, d. h. die Queue des Folgezustands ist immer identisch zur Queue vor Auftreten des Ereignisses. Nachfolgend werden jedoch neue Ereignisse für asynchrone Nachrichten vorgestellt, die den Queues Nachrichten hinzufügen, bzw. diese aus ihnen entfernen. Beide Arten von Ereignissen können nicht in Zeitentscheidungszuständen auftreten.

Der Zeitbedarf von Nachrichten, die sich in Übertragung (also in einer Queue) befinden, wird durch spezielle Uhren modelliert. Für jede gleichzeitig in Übertragung befindliche Nachricht wird eine Uhr zur Modellierung der bisherigen Übertragungsdauer benötigt. Diese Uhren haben Auswirkungen auf die Berechnung der initialen Zones aller SG-Zustände zwischen Senden und Empfangen der Nachricht. Da diese Uhren beim Auftreten von Sende- und Empfangsereignissen erzeugt bzw. entfernt werden, werden sie nachfolgend im Zusammenhang mit diesen Ereignissen erläutert.

Asynchrone Sendeereignisse

Das Senden asynchroner Nachrichten wird durch *asynchrone Sendeereignisse* repräsentiert. Diese treten auf, wenn sonst ein Nachrichtenereignis auftreten würde, aber in der MSD-Spezifikation für die betreffende Kommunikationsverbindung und in die betreffende Richtung festgelegt ist, dass sich mehr als 0 Nachrichten im Transit befinden können.

Ein asynchrones Sendeereignis kann nur dann auftreten, wenn im aktuellen Zustand in der Queue für die betreffende Kommunikationsverbindung noch genügend Platz ist. Dies bedeutet, dass sich weniger Nachrichten in der Queue befinden müssen, als der Spezifikation nach in Übertragung sein können. Weder Systemobjekte noch Umgebungsobjekte können über eine volle Queue Nachrichten senden.

Wie Nachrichtenereignisse können asynchrone Sendeereignisse kontrollierbar oder unkontrollierbar sein. Abhängig davon werden sie beim Weiterschalten des Cuts unterschiedlich behandelt: Kontrollierbare asynchrone Sendeereignisse führen zu einem sofortigen Weiterschalten des Cuts, werden also genauso behandelt wie Nachrichtenereignisse. Für die Erfüllung von Anforderungen ist also nur der Sendezeitpunkt direkt relevant, der Empfangszeitpunkt kann aber für die rechtzeitige Reaktion eines anderen Subsystems wichtig sein. Unkontrollierbare asynchrone Sendeereignisse verändern den Cut jedoch noch nicht – hier wird der Cut erst beim Empfangen der Nachricht weitergeschaltet. Diese Festlegung bewirkt, dass das System nicht auf Umgebungsnachrichten reagieren muss, die es noch gar nicht erhalten hat.

Im Folgezustand des Sendeereignisses wird die durch das Ereignis repräsentierte Nachricht der Warteschlange für die betreffende Kommunikationsverbindung hinzugefügt. Wie bei Nachrichtenereignissen kann vor asynchronen Sendeereignissen beliebig viel Zeit vergehen. Da sie selbst nur den Beginn der Nachrichtenübermittlung repräsentieren, benötigen asynchrone Sendeereignisse keine Zeit. Das Vergehen von Zeit wird erst durch das Empfangsereignis modelliert. Daher wird die initiale Zone des Folgezustands eines asynchronen Sendeereignisses grundsätzlich wie bei Nachrichtenereignissen berechnet.

Zusätzlich wird aber in der initialen Zone des Folgezustands eine neue Uhr erzeugt, die den Zeitabstand seit dem Senden der Nachricht modelliert und die daher zunächst den Wert 0 hat. In allen folgenden Zuständen bis zum Empfangen der Nachricht gilt für diese Uhr in jeder initialen Zone eine obere Grenze (ähnlich einer Invariante in Timed Automata) identisch zur maximalen Übertragungsdauer, die für den Konnektor definiert wurde.

Da die Werte aller Uhren mit gleicher Rate steigen, können auch andere Uhren seit dem Sendeereignis nur um maximal diesen Wert ansteigen. Wie im nächsten Abschnitt genauer erläutert wird, kann dadurch das Auftreten von Verzögerungsereignissen und von asynchronen Empfangsereignissen für andere Nachrichten unterbunden werden. Erst das passende asynchrone Empfangsereignis hebt diese Einschränkung wieder auf.

Asynchrone Empfangsereignisse

Das Empfangen asynchroner Nachrichten wird durch *asynchrone Empfangsereignisse* repräsentiert. Diese sind immer unkontrollierbar, jedoch gilt für sie die implizite Umgebungsannahme, dass sie nach der definierten minimalen Übertragungszeit und spätestens bei Erreichen der maximalen auftreten müssen – relativ zum Sendezeitpunkt. Dabei kann die Umgebung entscheiden, zu welchem konkreten Zeitpunkt dies geschieht.

Eine Voraussetzung für das Auftreten eines asynchronen Empfangsereignisses in einem SG-Zustand ist, dass eine entsprechende Nachricht am Anfang einer der Nachrichten-Queues des Zustands vorhanden ist. Zusätzlich gilt, dass das Auftreten eines asynchronen Empfangsereignisses das Vergehen von genügend Zeit impliziert, um die minimale

Übertragungsdauer zu erreichen, die für den betreffenden Konnektor definiert wurde – ähnlich wie bei Verzögerungsereignissen. Wie bereits im vorherigen Abschnitt beschrieben wurde, wird bei jedem asynchronen Senden einer Nachricht eine Uhr erzeugt, deren obere Schranke das Vergehen von mehr Zeit als der maximalen Übertragungsdauer verhindert. Asynchrone Empfangsereignisse, deren minimale Übertragungsdauer bei diesen Einschränkungen nicht erreicht werden kann, können deshalb (zunächst) nicht auftreten – da die anderen Nachrichten zwangsläufig früher ankommen.

Asynchrone Empfangsereignisse können auch Verzögerungsereignisse verhindern und umgekehrt. Dabei verhindert ein Verzögerungsereignis das Auftreten eines asynchronen Empfangsereignisses nur dann, wenn seine Zeitbedingung erfüllt sein muss bevor die Nachricht bei minimaler Übertragungszeit angekommen sein kann. Umgekehrt verhindert ein asynchrones Empfangsereignis das Auftreten eines Verzögerungsereignisses nur dann, wenn die Nachricht auch bei maximaler Übertragungszeit früher angekommen sein muss als die Zeitbedingung des Verzögerungsereignisses erfüllt ist. Für beide Überprüfungen wird die Funktion `earlier` aus Abschnitt 4.3.3 verwendet.

Ein asynchrones Empfangsereignis führt im Folgezustand dazu, dass die Nachricht, deren Empfang das Ereignis repräsentiert, aus der betreffenden Queue entfernt wird. Bei asynchronen Empfangsereignissen für unkontrollierbare Nachrichten wird anschließend der Cut aller MSDs wie bei einem Nachrichtenereignis für diese Nachricht modifiziert.

Zur Berechnung der initialen Zone des Folgezustands aus derjenigen des aktuellen SG-Zustands wird zunächst das Vergehen von mindestens der minimalen Übertragungszeit min bis höchstens der maximalen Übertragungszeit max der repräsentierten Nachricht modelliert. Hierzu werden für die Uhr c, die der Nachricht zugeordnet ist, eine untere Schranke $c \geq min$ und eine obere Schranke $c \leq max$ definiert. Der im aktuellen Zustand s durch Warten erreichbare Zeitbereich $initial(s)^{\uparrow}$ wird mit diesen beiden Schranken geschnitten.

Aus der resultieren Clock Zone wird die Uhr c dann wieder entfernt; auch ihre Schranken gelten dann nicht mehr. Schließlich werden Clock Resets angewendet, sofern diese beim Weiterschalten des Cuts angetroffen werden. Das Ergebnis ist dann die initiale Zone des Folgezustands. Obwohl die Uhr c aus der neuen initialen Zone wieder entfernt wird, ist das Schneiden der Clock Zone mit diesen Schranken notwendig, um zu modellieren, dass der Wert aller anderen Uhren zwischen Senden und Empfangen der Nachricht im Bereich min bis max gestiegen ist.

4.4.4 Erweiterung des Algorithmus

Der in Abschnitt 4.3 für zeitbehaftete Systeme erweiterte Algorithmus wird hier erneut erweitert, um die im vorherigen Abschnitt eingeführten asynchronen Sende- und Empfangsereignisse zu unterstützen. Für diese muss auch die Berechnung der unsicheren Zones gegenüber der Beschreibung in Abschnitt 4.3.3 entsprechend erweitert werden. Zusätzliche Anpassungen sind erforderlich, damit die neuen architekturbezogenen Übertragungszeiten auch für zusätzlich in das System eingefügte Synchronisationsnachrichten berücksichtigt werden, die nicht bereits in der MSD-Spezifikation definiert sind.

Um die Korrespondenzbeziehungen zwischen Controller- und SG-Zuständen nach der Definition in Abschnitt 3.3.3 berechnen zu können, muss für alle Ereignisse (und damit

für deren Transitionen) definiert sein, ob diese *beobachtbar* sind oder nicht. Daher wird hier die Beobachtbarkeit auch der asynchronen Sende- und Empfangsereignisse definiert:

- Ein kontrollierbares asynchrones Sendeereignis ist für einen Controller beobachtbar, wenn dieser die entsprechende Nachricht sendet.
- Ein asynchrones Empfangsereignis ist für einen Controller beobachtbar, wenn dieser die entsprechende Nachricht empfängt.
- Für alle anderen Kombinationen von Controllern und asynchronen Sende-/Empfangsereignissen ist das Ereignis für den Controller unbeobachtbar.

Dies modelliert, dass nur der Sender vom Senden einer Nachricht erfährt, solange diese nicht beim Empfänger angekommen ist. Hieraus ergibt sich insbesondere, dass unkontrollierbare asynchrone Sendeereignisse generell nicht beobachtet werden können, ebenso wie die Empfangsereignisse für an die Umgebung gesendete Nachrichten.

Der Algorithmus muss für die Implementierung von asynchronen Sende- und Empfangsereignissen so erweitert werden, dass er diese neuen Ereignistypen bei der Erzeugung neuer Kandidaten berücksichtigt. In der erweiterten Variante wird im neuen Kandidaten ein beobachtbares asynchrones Ereignis e in einem SG-Zustand s implementiert, indem passende Transitionen im sendenden bzw. empfangenden Controller in allen zu s korrespondierenden Zuständen angelegt werden: Wenn e ein asynchrones Sendeereignis ist, werden entsprechende Transitionen im sendenden Controller angelegt, andernfalls im empfangenden. Für nicht beobachtbare asynchrone Ereignisse werden keine Controllertransitionen angelegt.

Anschließend werden die neuen Korrespondenzen der Controllerzustände durch eine Tiefensuche im SG über nicht beobachtbare Transitionen berechnet, wie es bereits in Abschnitt 3.3.3 für die Behandlung von Nachrichtenereignissen beschrieben wurde. Dabei wird jedoch die in Abschnitt 2.2.1 eingeführte, in Abschnitt 4.2.4 und in diesem Abschnitt erweiterte Definition für Beobachtbarkeit angewendet.

Übertragungszeit von asynchronen Nachrichten

Für asynchrone Ereignisse gilt generell weiterhin die Definition der unsicheren Zones `unsafe(s)` für SG-Zustände ohne Zeitentscheidungsereignisse aus Abschnitt 4.3.3. Das jeweilige Ereignis wird also abhängig von dessen Kontrollierbarkeit behandelt.

Die unsicheren Zones in Zuständen mit asynchronen Empfangsereignissen können sich jedoch durch die Zeitverzögerung verschieben und durch die zusätzliche Unsicherheit bei der Übertragungszeit (Differenz zwischen maximaler und minimaler Übertragungszeit) vergrößern. Die Umgebung darf das Auftreten des Empfangsereignisses jedoch nicht durch zusätzliches Warten verzögern. Daher *entfällt* für diese Ereignisse der zusätzliche unsichere Bereich vor `uc_unsafe(s)`, also die letzte Zeile in Gleichung 4.7.

Von einem SG-Zustand s aus kann eine unsichere Zone in einem Folgezustand s' durch ein asynchrones Empfangsereignis e in einem Zeitraum erreicht werden, der sich aus der möglichen *verbleibenden* Übertragungszeit ergibt. Von der in der Spezifikation definierten minimalen bzw. maximalen Übertragungszeit (nachfolgend min_{def} bzw. max_{def})

muss also die in Zustand s bereits vergangene Übertragungszeit abgezogen werden. Diese entspricht dem Wert der Uhr c, die für die empfangene Nachricht angelegt wurde, in der initialen Zone von s. Auch dieser Wert kann zwischen einem Minimum (min_c) und einem Maximum (max_c) schwanken. Die minimal bzw. maximal verbleibende Übertragungszeit ergibt sich dann als $min = min_{def} - max_c$ bzw. $max = max_{def} - min_c$. Dabei gilt aber $max \geq min \geq 0$.

Innerhalb des Intervalls [min,max] für die mögliche verbleibende Übertragungszeit bestimmt die Umgebung die konkrete Übertragungszeit. Daher muss immer vom ungünstigsten Fall für das System ausgegangen werden. Tritt e in s bis zu max Zeiteinheiten vor *Beginn* der unsicheren Zone in s' auf, dann kann diese in s' noch erreicht werden. Hier ist also die maximale Übertragungszeit der ungünstigste Fall. Tritt e bis zu min Zeiteinheiten vor *Ende* der unsicheren Zone in s' auf, dann kann diese in s' ebenfalls noch erreicht werden – danach jedoch nicht mehr. Hier ist also die minimale Übertragungszeit der ungünstigste Fall.

Abbildung 4.18 zeigt ein Beispiel für die Verschiebung einer unsicheren Zone durch eine Nachricht `m1` mit einer Übertragungszeit von [3, 4] Zeiteinheiten. Die Uhr zu `m1` ist hier c_s1_s2. Da dem Uhrwert nach bei Betreten von s bereits 1 bis 2 Zeiteinheiten seit dem Senden vergangen sind, verbleiben mindestens eine (=3-2), maximal drei (=4-1) Zeiteinheiten bis `m1` empfangen wird und dabei s' erreicht wird. In s' ist eine unsichere Zone [3,5] definiert. Da die verbleibende Übertragungszeit im Intervall [1,3] liegt, verschiebt sich der Beginn der unsicheren Zone zu 0 (=3-3), das Ende zu 4 (=5-1).

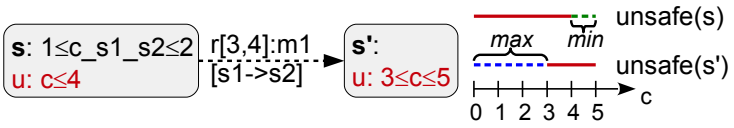


Abbildung 4.18: Verschiebung von unsicheren Zones durch verzögertes Empfangen

Die Definition der Hilfsfunktion `adapt(x,s,s')` (siehe Abschnitt 4.3.3) zur Berechnung der unsicheren Zone in s auf Basis der unsicheren Zone x in Folgezustand s' muss daher für asynchrone Empfangsereignisse angepasst werden. Dazu werden die unteren und oberen Schranken der nach bisheriger Definition von `adapt` erzeugten Clock Zone für eine in s empfangene Nachricht m, die zu s' führt, wie folgt modifiziert:

- Die *maximal* verbleibende Übertragungszeit für m wird von den *unteren* Schranken aller Uhren abgezogen. Dies modelliert, dass ein Verlassen von s zu s' durch Empfangen von m erst nach der maximalen Übertragungszeit garantiert ist, wodurch die neue unsichere Zone in s um diese Zeitspanne früher beginnt. Die Nachricht m *muss* also entsprechend früher gesendet werden.

- Die *minimal* verbleibende Übertragungszeit für m wird von den *oberen* Schranken aller Uhren abgezogen. Dies modelliert, dass ein Verlassen von s zu s' durch Empfangen von m erst nach der minimalen Übertragungszeit angenommen werden muss, wodurch die neue unsichere Zone in s um diese Zeitspanne früher endet. Die Nachricht m *darf* also entsprechend früher wieder gesendet werden.

Demnach ist eine unsichere Zone in s also gegenüber ihrer Entsprechung in s' um max Zeiteinheiten früher und um $max - min$ Zeiteinheiten länger. Beide Schranken werden anschließend aber auf mindestens 0 gesetzt, wodurch es effektiv also auch zu einer Verkürzung der unsicheren Zone in s kommen kann.

Übertragungszeit von Synchronisationsnachrichten

Auch das Übertragen von Synchronisationsnachrichten benötigt nun Zeit entsprechend der verwendeten Kommunikationsverbindung. Muss im aktuellen Controllersystem ein Controller vor dem Senden einer Nachricht eine Synchronisationsnachricht erhalten, so kann er erst nach Ablauf der Übertragungszeit dieser Nachricht senden. Dabei muss wieder vom ungünstigsten Fall, also der maximalen Dauer ausgegangen werden. Sind zur Synchronisation mehrere Nachrichten erforderlich (siehe Abschnitt 3.3.7), so muss das Maximum der benötigten (maximalen) Übertragungszeiten dieser Nachrichten angenommen werden. Bei Ketten von Synchronisationen (siehe Abschnitt 3.4) müssen diese Zeiten entsprechend aufaddiert werden.

Die zur Synchronisation benötigten Zeitdauern müssen im Synthesealgorithmus ebenfalls durch eine Modifikation der Berechnung von unsicheren Zones durch `unsafe` berücksichtigt werden. Dabei sind Synchronisationsnachrichten nur in Zuständen relevant, in denen das System eine Nachricht der Spezifikation sendet, also insbesondere nicht in Zeitentscheidungszuständen.

Der Zeitbedarf von Synchronisationen hat Auswirkungen auf die Berechnung der unsicheren Zones für diejenigen kontrollierbaren Transitionen, die eine Synchronisation benötigen. Diese Transitionen können nun nicht mehr das Schalten von unkontrollierbaren Transitionen in der initialen Zone des aktuellen Zustands verhindern: Der Zustand kann jederzeit innerhalb der initialen Zone betreten werden, das Schalten der kontrollierbaren Transition ist aber erst mit Verzögerung möglich, während gleichzeitig aktive unkontrollierbare Transitionen sofort schalten können.

Das tatsächliche Senden einer kontrollierbaren Nachricht kann erst garantiert werden, wenn *nach* der initialen Zone ein Zeitraum vergangen ist, welcher der maximalen Übertragungszeit der erforderlichen Synchronisationen entspricht. Daher muss die Definition der Hilfsfunktion `c_unsafe(s)` zur Berechnung der kontrollierbaren unsicheren Zones entsprechend angepasst werden. Die übrige Definition von `unsafe` kann jedoch unverändert bleiben.

Die neue Definition für `c_unsafe(s)` für einen SG-Zustand s ohne Zeitentscheidungsereignisse mit ausgehenden kontrollierbaren implementierten Transitionen T_c ist nun:

$$c\_unsafe(s) = \bigvee_{t \in T_c} successor\_unsafe(s,t) \vee initial(s)^{\uparrow[0,syncdelay(t)]} \tag{4.9}$$

Dabei liefert die Funktion `syncdelay(t)` die maximale Übertragungszeit der für das Schalten der kontrollierbaren Transition t benötigten Synchronisationsnachrichten.

Bei der Extraktion der zeitbehafteten Controller (vgl. Abschnitt 4.3.4) werden die Invarianten für Controllerzustände, die Nachrichten senden, wie bisher berechnet. Wenn jedoch Synchronisationen vor dem Senden erforderlich sind, dann müssen in den Controllerzuständen, die diese Synchronisationsnachrichten senden, jeweils engere Invarian-

ten angelegt werden: Von der Bedingung der ursprünglichen Invariante für das Senden einer Nachricht der Spezifikation wird jeweils die maximale Übertragungszeit aller noch zu sendenden Synchronisationsnachrichten subtrahiert. Es darf in den Zwischenzuständen also jeweils nur so lange gewartet werden, dass für die restlichen Synchronisationsnachrichten und für die anschließend gemäß der Spezifikation gesendete Nachricht noch genügend Zeit bleibt.

4.4.5 Behandlung des Beispielproblems

Die diskutierten Erweiterungen werden hier anhand der erweiterten MSD-Spezifikation aus Abschnitt 4.4.2 für das Anwendungsbeispiel der Produktionszelle illustriert.

Abbildung 4.19 zeigt den Zwischenstand der Synthese nachdem `cIB` $\leq$ `30` in S.11 implementiert wurde. Der Aufbau der Abbildung und die Bedeutung der Bildelemente entsprechen dem Beispiel in Abschnitt 4.3.6. Hier werden nur die Unterschiede zu diesem Beispiel erläutert.

Ein wesentlicher Unterschied zum Fall mit ausschließlich synchroner Kommunikation ist die separate Behandlung von Senden (Präfix „s:“) und Empfangen (Präfix „r[min,max]:“) der asynchronen Nachrichten im SG. Dabei kann der Sender einer Nachricht nur das Sendeereignis beobachten, der Empfänger nur das Empfangsereignis. Im Fall der in S.4 gesendeten Nachricht `blankAtPress` ist für aC also nur der Übergang zu S.5 sichtbar, aber nicht der von S.5 zu S.6, während pC nur letzteres beobachten kann. Entsprechend werden die Korrespondenzen der Controllerzustände berechnet bzw. neue Controllerzustände angelegt. Das Senden der Umgebungsnachricht `done` in S.9 ist für das System grundsätzlich nicht feststellbar; bei allen an die Umgebung gesendeten Nachrichten (z. B. `press` in S.6) gilt dasselbe für das Empfangen.

In allen Spezifikationszuständen, die durch ein asynchrones Sendeereignis erreicht werden, wird eine zusätzliche temporäre Uhr erzeugt und mit 0 initialisiert. Diese misst die vergangene Zeit seit dem Senden der Nachricht. Dies ist hier in S.5, S.7 und S.10 der Fall. Der Name dieser Uhr hat hier die Form `c_<Sender>_<Empfänger>`. Beispielsweise modelliert c_aC_pC die Übertragungszeit für `blankAtPress` in S.5.

In allen SG-Zuständen, die durch ein asynchrones Empfangsereignis erreicht werden, wird die minimale Verzögerung des Ereignisses auf die unteren Schranken der initialen Zone addiert, während die maximale Verzögerung auf die obere Schranke addiert wird. Beispielsweise gilt in S.5 initial (unter anderem) `cPress = 0` $\wedge$ `cIB` $\geq$ `3`. Daher hat S.6 nach einer Verzögerung im Intervall [2,3] die initiale Zone `cPress` $\geq$ `2` $\wedge$ `cPress` $\leq$ `3` $\wedge$ `cIB` $\geq$ `5` (cBtP entfällt durch Terminierung eines MSDs).

Nachfolgend wird erläutert, wie sich die Berechnung der gezeigten unsicheren (Präfix „u:“) und ausgeschlossenen (Präfix „e:“) Zones ändert. Die Entstehung der Zones in den Zuständen S.11 und S.12 ändert sich durch die asynchrone Kommunikation nicht. Auch die Weiterpropagierung zu den Vorgängerzuständen funktioniert prinzipiell gleich. Dabei können sich die Zones jedoch, wie in Abschnitt 4.4.4 erläutert wurde, verschieben.

In Zustand S.10 werden die oberen und unteren Schranken der unsicheren und ausgeschlossenen Zones aus S.11 für alle Uhren um eine Sekunde reduziert. Dies modelliert, dass von S.10 zu S.11 durch die Übertragung von `done` genau eine Sekunde vergeht.

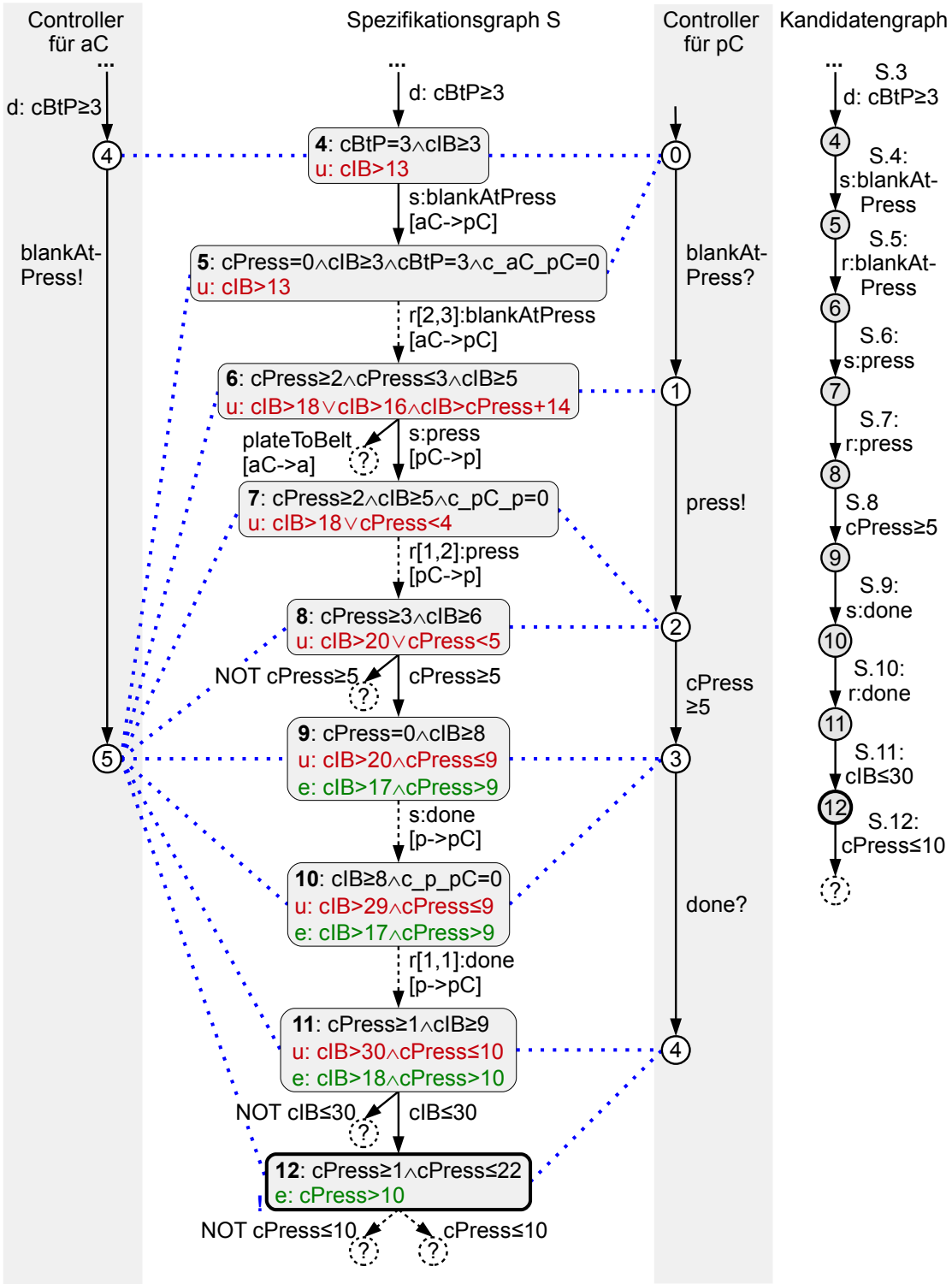


Abbildung 4.19: Zwischenstand für das Beispiel mit asynchroner Kommunikation

Die Berechnung der unsicheren Zones für S.9 und S.8 entspricht dem bereits beschriebenen Fall mit synchroner Kommunikation. In Zustand S.7 wird, entsprechend der Verzögerung [1,2], die obere Schranke um eine Sekunde, die untere Schranke um zwei Sekunden reduziert, um den schlechtesten Fall anzunehmen. Das System muss also ein Verlassen dieses Zustands bereits bis `cIB = 18` sicherstellen, da bis zum Erreichen von S.8 bis zu zwei Sekunden vergehen können. Andererseits darf es aber eine Sekunde früher senden, da S.8 erst frühestens eine Sekunde später erreicht wird.

Die unsicheren Zones in S.6 modellieren, dass in S.6 ein Warten bis `cPress = 4` möglich sein muss, um die unsichere Zone `cPress < 4` in S.7 zu vermeiden. Da immer der ungünstigste Fall angenommen werden muss, muss in S.6 cPress = 2 angenommen und weitere zwei Sekunden gewartet werden – auch wenn bereits cPress = 3 gilt. Die zweite unsichere Zone in S.6 modelliert daher, dass bereits für `cIB > 16` ein Aufenthalt in S.6 mit einer zu hohen Differenz von cIB zu cPress unsicher ist, weil dann kein ausreichend langes Warten mehr möglich ist. Die andere unsichere Zone modelliert, dass in keinem Fall die aus S.7 übernommene unsichere Zone `cIB > 18` erreicht werden darf.

Aufgrund der maximalen Verzögerung von 3 Sekunden für das Senden von `blankAtPress` muss S.5 spätestens bei `cIB = 13` verlassen werden, da sonst in S.6 die zweite unsichere Zone erreicht wird: Bei Betreten von S.6 mit `cIB > 16` gilt `cIB > cPress+14` durch `cPress` $\geq$ `2` in der initialen Zone zwangsläufig. Um S.5 vor der unsicheren Zone zu erreichen, muss in S.4 `blankAtPress` bis `cIB = 13` gesendet werden. Die Berechnung der unsicheren Zones für die übrigen (nicht gezeigten) Vorgängerzustände ändert sich nicht gegenüber dem Fall mit synchroner Kommunikation.

4.5 Zusammenfassung

In diesem Kapitel wurde der in Kapitel 3 vorgestellte Synthesealgorithmus für Echtzeitsysteme angepasst. Dabei wurden in einem ersten Schritt zunächst Spezifikationsgraphen (SG) auf Basis zeitbehafteter MSD-Spezifikationen (siehe Abschnitt 2.1.5) unterstützt. Anschließend wurde der Algorithmus so angepasst, dass dieser für eine gegebene zeitbehaftete MSD-Spezifikation für jedes Systemobjekt einen zeitbehafteten Controller erzeugt – ein auf Timed Automata basierendes Verhaltensmodell (siehe Abschnitt 2.2.2).

Der vorgestellte Ansatz basiert auf *Clock Zones* [Dil90, BY04], die Echtzeitverhalten symbolisch repräsentieren (siehe Abschnitt 2.2.2). Clock Zones wurden bereits zur Synthese globaler Controller eingesetzt [MPS95]. Der hier beschriebene Ansatz kombiniert dieses Konzept jedoch erstmals mit einem verteilten Synthesealgorithmus.

Der beschriebene Syntheseansatz unterstützt zeitbehaftete MSDs [Gre11] (die weitgehend den zeitbehafteten LSCs [HM02b, HM03] entsprechen). Für diese werden jedoch oft zwei unrealistische Annahmen vorausgesetzt: Zum einen wird von vernachlässigbar kurzen Übertragungszeiten von Nachrichten ausgegangen; zum anderen wird angenommen, dass das Senden und Empfangen jeder Nachricht in einem einzigen atomaren Schritt erfolgt, also keine Überlappung mit anderen Nachrichten möglich ist. Um den Syntheseansatz auch ohne Gültigkeit dieser Annahmen anwenden zu können, wurde das Verfahren in einem zweiten Schritt um die Modellierung und Berücksichtigung des Zeitbedarfs von Nachrichten und um ein asynchrones Kommunikationsmodell erweitert.

Die Implementierung der Konzepte dieses Kapitels wird in Abschnitt 6.1 vorgestellt, die Evaluierung anhand der Ausführung auf Beispiele wird in Abschnitt 6.2 behandelt.

Die als Ausgabe des Verfahrens erzeugten Timed Automata können prinzipiell bereits als Implementierungsmodell für verteilte mechatronische Systeme verwendet werden. Sie müssen jedoch oft um weitere Verhaltensbestandteile, beispielsweise kontinuierliches Verhalten, ergänzt werden. Zudem ist oft eine spezifischere manuelle Verfeinerung der Modelle für den konkreten Einsatzbereich oder für eine bestimmte Plattform erforderlich. Daher betrachtet das folgende Kapitel 5 die Verwendung des zeitbehafteten Syntheseverfahrens im Rahmen einer modellbasierten Entwicklungsmethode, die eine entsprechende Weiterentwicklung der Implementierungsmodelle ermöglicht.

Kapitel 5

Verteilte Synthese für mechatronische Systeme

Dieses Kapitel behandelt die Integration des in Kapitel 3 vorgestellten und in Kapitel 4 für Echtzeitsysteme erweiterten Syntheseansatzes in eine bestehende Entwicklungsmethode für die modellbasierte Entwicklung der Software mechatronischer Systeme. Dies hat das Ziel, die Weiterentwicklung der durch die Synthese erzeugten verteilten Implementierungsmodelle zur fertigen Software zu unterstützen. Dazu wird zum einen beschrieben, wie die Ein- und Ausgabemodelle des Syntheseverfahrens transformiert werden müssen, um dieses im Rahmen der Methode verwenden zu können. Zum anderen wird diskutiert, wie dazu der Entwicklungsprozess der Methode angepasst werden muss, insbesondere um die Korrektheit der entwickelten Software trotz manueller Anpassungen sicherzustellen.

Die als Eingabe der Synthese verwendete Spezifikationssprache der Modal Sequence Diagrams (MSDs) ist primär auf die Spezifikation nachrichtenbasierter Kommunikation ausgelegt, während kontinuierliches Verhalten nicht unterstützt wird und beispielsweise komplexe arithmetische Berechnungen oder Zeichenkettenoperationen nur eingeschränkt ausgedrückt werden können. Dazu kommt, dass die Verwendung von Variablen in MSD-Spezifikationen zu einer Zustandsraumexplosion bei der verteilten Synthese führen kann, da im Spezifikationsgraphen jede erreichbare Kombination von Variablenwerten durch mindestens einen Zustand repräsentiert wird. Es kann daher sinnvoll oder erforderlich sein, bestimmte Verhaltensbestandteile erst nach der Synthese manuell zu ergänzen. In jedem Fall sind nach der Synthese weitere Schritte erforderlich, um die generierten Controller durch plattformspezifische Informationen zu erweitern und schließlich aus ihnen Code zu generieren.

Das Syntheseverfahren muss also in einer Entwicklungsmethode verwendet werden, mittels der Verhaltensanteile ergänzt werden können, die durch die Synthese nicht erzeugt werden. Eine solche Entwicklungsmethode ist MechatronicUML [BDG+14a, BDG+14b, GS13]. MechatronicUML definiert einen Entwicklungsprozess für mechatronische Systeme und eine Sprache zur Modellierung von Struktur und Verhalten dieser Systeme. Bei der Verwendung der Methode werden Entwickler durch eine spezielle Entwicklungsumgebung, die MechatronicUML Tool Suite [DGB+14], unterstützt. Die Methode beinhaltet verschiedene Verifikationsverfahren, die – teils nach Anpassung zur Verwendung mit MSD-Spezifikationen – sicherstellen können, dass die Spezifikation nach manuellen Änderungen der synthetisierten Modelle noch eingehalten wird. Die Integration in diesen Entwicklungsansatz ermöglicht es somit, die durch die Synthese erzeugten verteilten Implementierungsmodelle zur fertigen Software weiterzuentwickeln.

Das Kapitel ist wie folgt aufgeteilt: Abschnitt 5.1 beschreibt zunächst, welche Modelle der MechatronicUML aus dem durch die verteilte Synthese erzeugten (zeitbe-

hafteten) Controllersystem abgeleitet werden können. Zudem wird diskutiert, wie die gegebene MSD-Spezifikation geändert werden muss, um Modelle zu erhalten, die in MechatronicUML besser weiterentwickelt werden können. MechatronicUML definiert einen Entwicklungsprozess, dessen Anpassung zur Integration der Synthese in Abschnitt 5.2 behandelt wird. Schließlich fasst Abschnitt 5.3 das Kapitel zusammen.

5.1 Erzeugung von MechatronicUML-Modellen aus synthetisierten Controllersystemen

Eine zentrale Frage bei der Integration des Syntheseansatzes ist, welche Bestandteile eines MechatronicUML-Modells aus den erzeugten Controllern abgeleitet werden können. Von dieser Frage hängt auch ab, welche Anpassungen am Entwicklungsprozess vorgenommen werden müssen.

In MechatronicUML werden Systeme komponentenbasiert entwickelt. Die zu entwickelnden Subsysteme und die Bestandteile der Umgebung werden also durch *Komponenten* modelliert. Diese können aus der Struktur der MSD-Spezifikation abgeleitet werden.

Grundsätzlich wird das Kommunikationsverhalten der entwickelten Systeme in MechatronicUML durch *Real-Time Statecharts (RTSCs)* [GB03] modelliert. Deren Semantik ist, wie die der zeitbehafteten Controller (siehe Abschnitt 2.2.2), durch Timed Automata definiert. Daher ist eine Konvertierung der (zeitbehafteten) Controller zu RTSCs naheliegend und relativ einfach möglich. In MechatronicUML werden RTSCs jedoch zu verschiedenen Zwecken verwendet und beschreiben dabei unterschiedliche Aspekte des Systemverhaltens auf unterschiedlichen Ebenen. Durch geeignete Anpassung der MSD-Spezifikation, also der Eingabe des Syntheseverfahrens, können unterschiedliche Anteile des Systemverhaltens erzeugt werden, die unterschiedlichen Typen von RTSCs entsprechen.

Abschnitt 5.1.1 beschreibt die Erzeugung von RTSCs aus den synthetisierten zeitbehafteten Controllern. Abschnitt 5.1.2 diskutiert, wie das Syntheseverfahren ohne Anpassungen zur Erzeugung von MechatronicUML-Modellen genutzt werden kann und welche weiteren Modellelemente neben RTSCs aus dem Syntheseergebnis oder der Spezifikation abgeleitet werden können. Dabei wird motiviert, dass für eine Synthese von Modellen, die im Kontext der Methode MechatronicUML sinnvoll weiterentwickelt werden können, zuvor eine Anpassung der MSD-Spezifikation durchgeführt werden sollte. Diese wird in Abschnitt 5.1.3 näher erläutert.

5.1.1 Umsetzung von Controllern als Real-Time Statecharts

Die Semantik von RTSCs wurde ursprünglich basierend auf der Semantik der *Extended Hierarchical Timed Automata (ExHTA)* [DM01] definiert [GB03]. ExHTA stellen eine Erweiterung der Timed Automata [Dil90, AD94] um Zustandshierarchien dar. Diese hierarchischen Zustände ähneln dabei denen der von Harel [Har87] eingeführten *Statecharts*, die später in Form der *State Machines* in die UML [Obj11] übernommen wurden.

Da die (zeitbehafteten) Controller ausschließlich einfache, nicht-hierarchische Zustände beinhalten, kann bei ihrer Abbildung auf RTSCs auf die Erzeugung hierarchischer Zustände verzichtet werden. Diese könnten zwar das Modell kompakter und besser lesbar machen; ihre automatische Ableitung wäre jedoch nicht trivial und wurde in dieser Arbeit nicht betrachtet, da dieses Problem nicht im Fokus der Arbeit liegt.

Beim Zusammenfügen mehrerer für eine Komponente erzeugten RTSCs zu einem einzigen werden diese in MechatronicUML üblicherweise in nebenläufigen *Regionen* eines *orthogonalen komplexen Zustands* – eines speziellen hierarchischen Zustands – kombiniert. Dieses Zusammenfügen wird jedoch nicht durch den Syntheseansatz automatisiert und verbleibt daher als manueller Schritt.

Da die zeitbehafteten Controller im Wesentlichen den Timed Automata entsprechen (vgl. Abschnitt 2.2.2), genügt es für die Konstruktion eines äquivalenten RTSCs, nur diejenigen Modellelemente zu verwenden, die bereits für Timed Automata definiert sind. Die Abbildung einzelner zeitbehafteter Controller auf die entsprechenden RTSCs ist daher eine einfache eins-zu-eins-Abbildung der Modellelemente der Controller auf die entsprechenden der RTSCs. Eine solche Modelltransformation wurde in dieser Arbeit umgesetzt (siehe Abschnitt 6.1). Es gibt nur die folgenden Spezialfälle: Beim Senden oder Empfangen von Nachrichten durch Transitionen muss in RTSCs syntaktisch zwischen synchroner und asynchroner Kommunikation unterschieden werden. Für erstere müssen Synchronisationskanäle erzeugt werden. Für letztere sind Einträge für die jeweiligen Nachrichten in einem „Message Type Repository“ erforderlich. Auch für die lokalen Uhren der RTSCs müssen entsprechende Deklarationen erzeugt werden.

Ein grundsätzlicher Unterschied zwischen Controllern und RTSCs ist, dass letztere in MechatronicUML oft das Verhalten mehrerer Instanzen eines Typs beschreiben, während die Synthese für jedes modellierte Objekt (also jede Instanz) einen eigenen Controller erzeugt. Je nach gegebener MSD-Spezifikation können die Anforderungen an mehrere Objekte aber durch denselben Controller erfüllt werden. Dies ist beispielsweise dann der Fall, wenn die betreffenden MSDs symbolische Lifelines verwenden, die jeweils mehrere Objekte repräsentieren können (siehe Abschnitt 2.1.4). Die durch die Synthese erzeugten Controller definieren dann Verhalten, das für alle diese Objekte korrekt ist (oder sie sind sogar identisch) und können daher gegeneinander ausgetauscht werden. In diesem Fall können die redundant erzeugten Controller nach der Synthese manuell gelöscht werden. Nur aus den verbleibenden müssen dann RTSCs erzeugt werden.

5.1.2 Anwendung der Synthese ohne Anpassungen

Bei der Modellierung der Anforderungen als MSD-Spezifikation wird normalerweise die Kommunikation zwischen eigenständigen Objekten beschrieben (siehe Abschnitt 2.1). Die erzeugten Controller beschreiben also jeweils das vollständige Kommunikationsverhalten eines Objekts. Die Objekte der MSD-Spezifikation entsprechen in MechatronicUML den Instanzen *atomarer Komponenten*, d. h. von Komponenten ohne Subkomponenten. Das Verhalten von Komponenten wird durch jeweils ein Real-Time Statechart beschrieben, das als *Komponenten-RTSC* bezeichnet wird. Die erzeugten Controller können also nach der Umwandlung in RTSCs direkt als Komponenten-RTSCs verwendet werden. Diese können dann Grundlage für Simulation und Codegenerierung sein.

Abbildung 5.1 zeigt die direkte Anwendung des Syntheseverfahrens zur Erzeugung von MechatronicUML-Komponenten anhand des Anwendungsbeispiels der Produktionszelle: Für jedes Objekt wird eine atomare Komponente erzeugt. Für jedes Systemobjekt (hier der ArmController aC und PressController pC) erzeugt die Synthese einen Controller, aus dem ein Komponenten-RTSC für die zugehörige Komponente erzeugt wird. Für jede Kommunikation einer Komponente mit einer anderen Komponente wird ein *Port* erzeugt, über den diese Kommunikation stattfindet. Die schwarzen Dreiecke in der Darstellung der Ports definieren die möglichen Kommunikationsrichtungen. Aus der in der MSD-Spezifikation definierten Struktur können zudem die initial im System existierenden *Komponenteninstanzen* mit entsprechenden *Portinstanzen* abgeleitet werden (unten im Bild). Zur Kommunikation der Komponenteninstanzen untereinander werden *Konnektoren* zwischen den betreffenden Portinstanzen angelegt. Die resultierende Struktur ist dieselbe, wie sie durch die Kompositionsstrukturdiagramme der MSD-Spezifikation definiert wird.

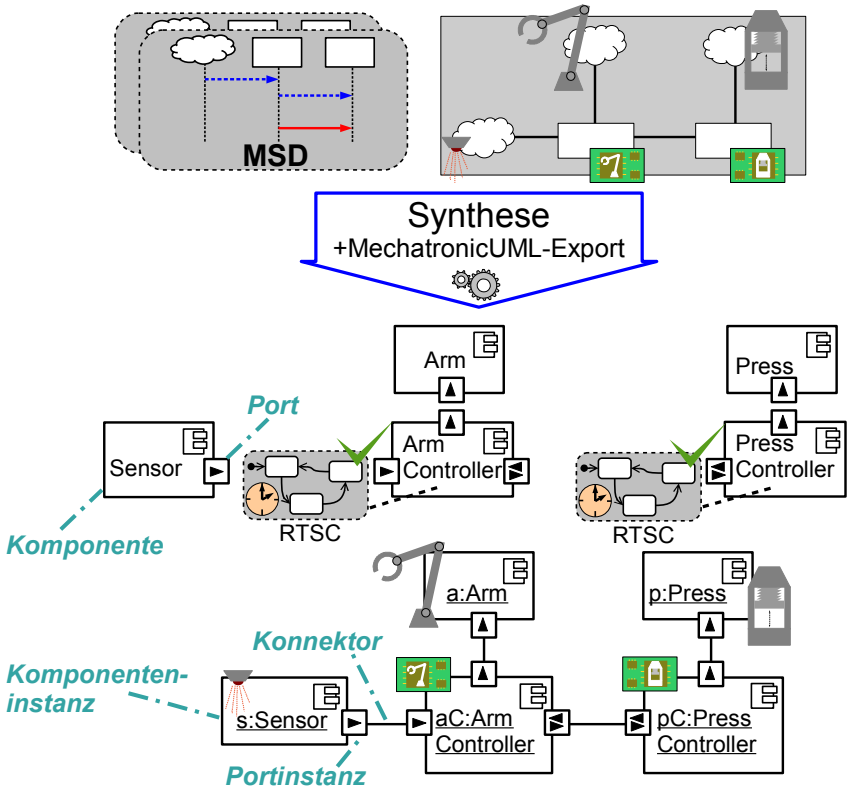


Abbildung 5.1: Direkte Anwendung des Syntheseverfahrens auf die Spezifikation

In MechatronicUML werden Komponenten-RTSCs jedoch normalerweise nicht von Grund auf neu erzeugt, sondern sie ergeben sich aus anderen RTSCs, die in früheren Entwicklungsschritten erstellt werden. MechatronicUML sieht für die Modellierung des Kommunikationsverhaltens einen mehrstufigen Prozess vor: Zunächst wird das Kommunikationsverhalten zwischen je zwei (Typen von) Subsystemen durch *Echtzeitkoordi-*

nationsprotokolle beschrieben, die sich zunächst noch nicht auf konkrete Komponenten beziehen. In diesen werden die Kommunikationspartner durch *Rollen* repräsentiert, deren Verhalten durch RTSCs, die *Rollen-RTSCs*, beschrieben wird. Abbildung 5.2 zeigt (oben) ein Echtzeitkoordinationsprotokoll mit dem Namen `PressCoordination` für das Beispiel.

Rollen beschreiben die Kommunikationspartner und ihr Verhalten zunächst abstrakt. Sie werden durch konkrete Ports von Komponenten implementiert, mit jeweils einem eigenen RTSC, einem *Port-RTSC* (vgl. Abbildung 5.2). Während die Rollen-RTSCs nur die Kommunikation mit der jeweils anderen Rolle beschreiben, enthalten die Port-RTSCs zusätzlich internes Komponentenverhalten, einschließlich der Abstimmung mit anderen Ports der Komponente. Das Verhalten eines Ports muss das der jeweiligen Rolle *verfeinern*, d. h. das Kommunikationsverhalten des Ports darf sich nur innerhalb spezieller Grenzen gegenüber dem der Rolle ändern. Das Komponenten-RTSC wird erst durch Kombination der Port-RTSCs gebildet, ergänzt um weiteres komponenteninternes Verhalten.

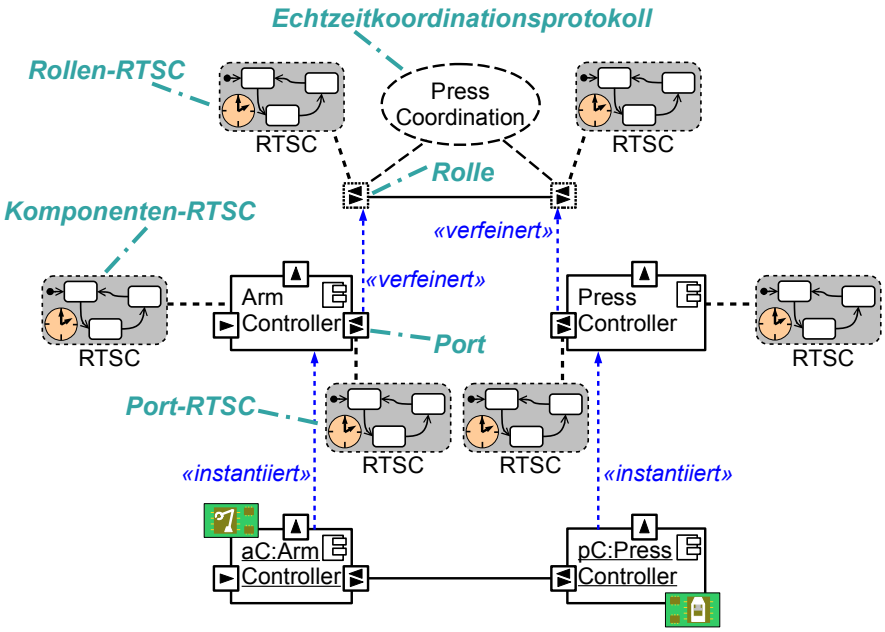


Abbildung 5.2: Überblick über die wichtigsten MechatronicUML-Syntaxelemente

Eine Entwicklung der Komponenten-RTSCs ohne vorherige Definition des Rollen- und Portverhaltens läuft der üblichen MechatronicUML-Entwicklungsmethode also entgegen. Eine direkte Erzeugung von Komponenten-RTSCs durch die Synthese ist nur dann sinnvoll, wenn die weitere Entwicklung des Systems nicht strikt dieser Entwicklungsmethode folgen soll – oder wenn die erzeugten Modelle nicht geändert oder aufgrund zusätzlicher Anforderungen verifiziert werden müssen. Beispielsweise ermöglicht MechatronicUML das Spezifizieren zusätzlicher, auf Zustände der RTSCs bezogener, Anforderungen an das Kommunikationsverhalten des Systems, die dann per *Model Checking* automatisch überprüft werden können. Im speziellen kompositionalen Verifika-

tionsansatz von MechatronicUML beziehen sich diese jedoch auf das Rollenverhalten und setzen dieses daher voraus.

Prinzipiell ist es auch nachträglich möglich, Rollen-RTSCs und Port-RTSCs aus Komponenten-RTSCs abzuleiten. Wenn keine besonderen Anforderungen an diese bestehen, dann können sie auf sehr einfache Weise wie folgt erzeugt werden: Für jede Kommunikation mit einer anderen Komponente wird ein Port mit einem trivialen Port-RTSC angelegt, das nur einen Zustand beinhaltet. Selbsttransitionen dieses Zustands für jede mögliche ein- oder ausgehende Nachricht leiten dann jede Kommunikation des generierten Komponenten-RTSCs weiter. Das zugehörige Rollen-RTSC kann dann durch Kopieren dieses Port-RTSCs erzeugt werden.

Die eigentliche Aufgabe von Port- und Rollen-RTSCs ist jedoch die möglichst vollständige Kapselung des Protokolls für die Kommunikation mit einem anderen Port bzw. einer anderen Rolle. Triviale RTSCs, die sehr wenig Verhalten beschreiben, sind daher für eine Weiterentwicklung mittels MechatronicUML kaum brauchbar.

5.1.3 Anpassung der MSD-Spezifikation vor der Synthese

Wird die Synthese auf eine geeignete MSD-Spezifikation angewendet, so kann sie direkt Controller erzeugen, welche nicht-trivialen Port- und Rollen-RTSCs entsprechen. Dazu muss die MSD-Spezifikation explizit die Kommunikation zwischen den einzelnen Ports oder Rollen der Komponenten modellieren. Objekte entsprechen dann nicht Komponenten, sondern eben den Rollen von Echtzeitkoordinationsprotokollen oder den Ports von Komponenten.

Komponenten haben jedoch normalerweise mehr als einen Port. Daher ist eine MSD-Spezifikation für Portverhalten typischerweise größer als eine für Komponentenverhalten, d. h. sie enthält mehr Modellelemente. Dies kann die Verständlichkeit verringern und den Modellierungsaufwand erhöhen. Alternativ kann zunächst eine MSD-Spezifikation auf Komponentenebene erstellt werden, von der dann nachträglich eine an MechatronicUML angepasste MSD-Spezifikation auf Port- oder Rollenebene abgeleitet wird. In diesem Abschnitt wird eine systematische Ableitung einer solchen angepassten Spezifikation beschrieben.

Zunächst werden die Anforderungen an eine solche Ableitung einer angepassten Spezifikation, sowie die Vor- und Nachteile diskutiert. Anschließend wird die systematische Durchführung dieser Ableitung beschrieben. Für diese wurde aus Zeitgründen im Rahmen dieser Arbeit keine Werkzeugunterstützung implementiert. Sie ist jedoch grundsätzlich automatisierbar.

Diskussion

Zunächst muss entschieden werden, ob die aus der gegebenen MSD-Spezifikation auf Komponentenebene abgeleitete neue Spezifikation durch ihre Lifelines Rollen oder Ports repräsentiert. In MechatronicUML ist vorgesehen, dass das abstrakte Kommunikationsverhalten des Systems durch die Rollen-RTSCs von Echtzeitkoordinationsprotokollen definiert wird. Dabei wird jedoch in jedem Echtzeitkoordinationsprotokoll (zumindest in der aktuellen Version von MechatronicUML [BDG+14b]) nur das Kommunikati-

onsverhalten zwischen *zwei* Rollen modelliert, die jeweils einen Port abstrahieren. Eine dieser Rollen kann dabei mehrfach instanziiert sein (eine „Multi-Rolle“), das RTSC ist jedoch in jeder Instanz dasselbe. Der Zusammenhang zwischen mehreren Echtzeitkoordinationsprotokollen wird erst in den konkreten Komponenten hergestellt, deren Ports die Rollen der Protokolle instanziieren.

Hängt beispielsweise die Kommunikation zwischen zwei Subsystemen von einem dritten ab, dann ist das in MechatronicUML erst auf Portebene modellierbar, während diese Abhängigkeit in den Rollen-RTSCs fehlt. Daraus ergibt sich, dass es im Allgemeinen nicht möglich ist, das gesamte Kommunikationsverhalten eines Systems nur auf Rollenebene zu spezifizieren. Eine Implementierung des gesamten Kommunikationsverhaltens einer MSD-Spezifikation kann daher in MechatronicUML nur auf Ebene von Ports erfolgen. Daher müssen sich die Lifelines der abgeleiteten MSD-Spezifikation auf Ports statt auf Rollen beziehen.

Bei der Kommunikation zwischen Ports muss in MechatronicUML unterschieden werden, ob es sich um Ports derselben Komponente oder um Ports verschiedener Komponenten handelt. Im ersten Fall kann nur synchron, im zweiten Fall nur asynchron kommuniziert werden [BDG+14b]. Die abgeleitete MSD-Spezifikation muss daher ein Architekturmodell enthalten, das diese Einschränkung berücksichtigt.

Ein Vorteil der Synthese aus einer MSD-Spezifikation auf Portebene ist, dass die so erzeugten Port-RTSCs typischerweise einen deutlich größeren Anteil des Verhaltens für die Kommunikation mit einer anderen Komponente enthalten, als bei der in Abschnitt 5.1.2 skizzierten Ableitung trivialer Port-RTSCs aus einem Komponenten-RTSC. Dies wird durch den Syntheseansatz jedoch nicht garantiert. Zumindest aber lassen sich aus den durch die Synthese erzeugten Controllern direkt Port-RTSCs erzeugen. Es ist also – anders als bei der Synthese für MSD-Spezifikationen auf Komponentenebene – kein weiterer Ansatz zum nachträglichen Aufteilen des Verhaltens erforderlich.

Ein Nachteil ist eine Erhöhung der Laufzeit und des Speicherbedarfs der Synthese durch die zusätzlichen Modellelemente. Insbesondere bei einer hohen Anzahl von Ports pro Komponente steigt die Anzahl der Objekte in der Spezifikation stark an. Da aber ein nachträgliches Aufteilen von synthetisierten Komponenten-RTSCs ebenfalls aufwendig sein kann, ist der Gesamtaufwand nicht unbedingt höher als in der direkten Variante, wenn in beiden Fällen als Endergebnis Port-RTSCs vorliegen sollen.

Im Falle einer manuellen Ableitung einer MSD-Spezifikation auf Portebene aus einer MSD-Spezifikation auf Komponentenebene ist der hohe Aufwand für die Entwickler ein weiterer Nachteil. Dazu kommt, dass die Entwickler die Einhaltung der oben erwähnten Einschränkungen der Kommunikation gemäß dem MechatronicUML-Ansatz manuell sicherstellen müssen. Die Entwickler können bei der Ableitung zudem Fehler in die neue Spezifikation einführen. Diese Nachteile können jedoch durch eine Automatisierung der Ableitung vermieden werden. Die Entwickler müssen die automatisch generierte Spezifikation dann nicht mehr manuell anpassen oder im Einzelnen verstehen.

Systematische Ableitung

Dieser Abschnitt beschreibt genauer, wie eine für MechatronicUML angepasste MSD-Spezifikation auf Portebene aus einer gegebenen MSD-Spezifikation auf Komponenten-

ebene abgeleitet werden kann. Abbildung 5.3 zeigt diese Ableitung schematisch. Auf diese Abbildung wird nachfolgend Bezug genommen.

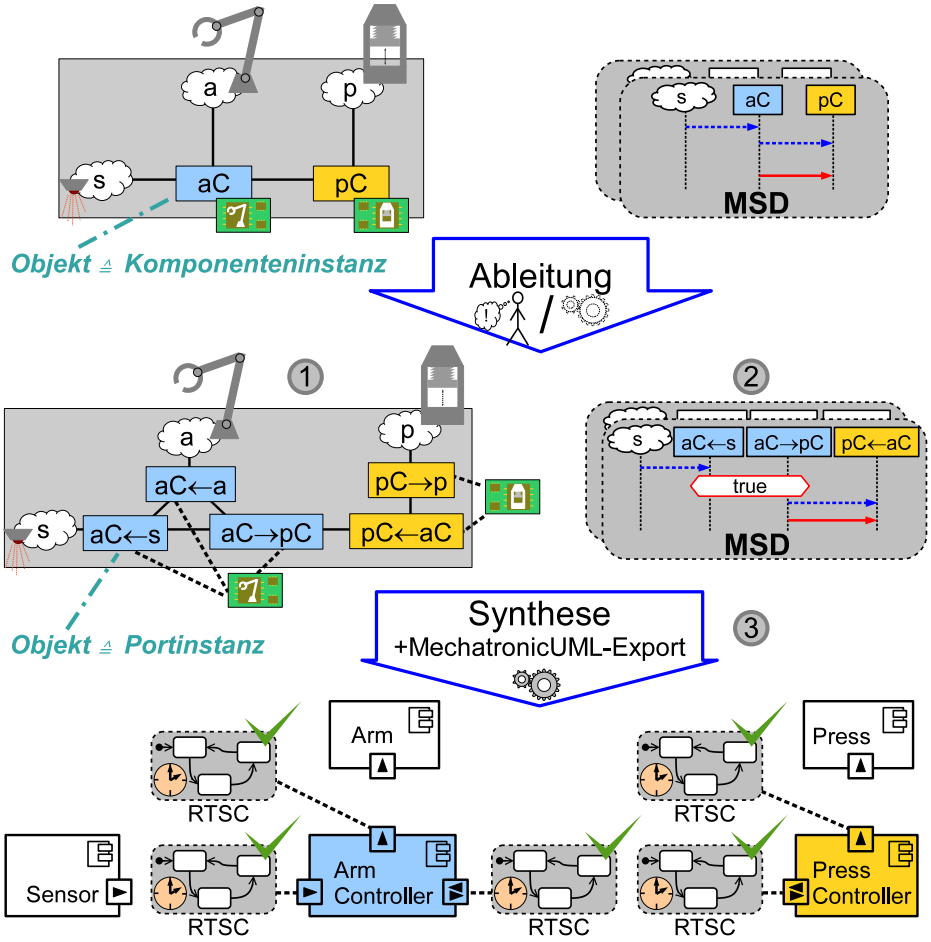


Abbildung 5.3: Anwendung des Syntheseverfahrens nach Anpassung der Spezifikation

Für jede Kommunikation einer Komponente mit einer anderen Komponente ist jeweils mindestens ein Port erforderlich. Daher werden für jede Kombination kommunizierender Objekte in der ursprünglichen MSD-Spezifikation Objekte erzeugt, die die beteiligten Portinstanzen repräsentieren **(1)**. Im Beispiel wird aC deshalb zu drei Objekten, pC zu zwei Objekten. Der in der gegebenen Spezifikation definierte Konnektor für die jeweilige Kommunikation wird entsprechend übernommen. Zwischen allen Objekten, die Portinstanzen derselben Komponenteninstanz repräsentieren, werden Konnektoren für synchrone Kommunikation angelegt – diese Objekte dürfen also uneingeschränkt und ohne (bzw. mit als vernachlässigbar angenommenem) Zeitaufwand miteinander kommunizieren.

Zusätzlich werden neue Klassen aus denen der ursprünglichen MSD-Spezifikation abgeleitet: Die Klassen der gegebenen Spezifikation repräsentieren jeweils Komponenten, während die Klassen der neuen Spezifikation Ports repräsentieren. Aus jeder Klasse auf

Komponentenebene wird eine Menge von Klassen auf Portebene gebildet – für jeden Port der Komponente eine. Die Methoden der ursprünglichen Klasse werden dabei so aufgeteilt, dass jede Port-Klasse diejenigen Methoden erhält, die den über den jeweiligen Port empfangenen Nachrichten entsprechen.

Alle Nachrichten, die in der ursprünglichen MSD-Spezifikation zwischen zwei Objekten (≙Komponenteninstanzen) ausgetauscht werden, werden in der neuen Spezifikation zwischen den beiden Objekten ausgetauscht, welche die Portinstanzen zur Kommunikation mit der jeweils anderen Komponenteninstanz repräsentieren.

Jede Lifeline, die in der alten Spezifikation eine Komponenteninstanz repräsentierte, wird in der neuen Spezifikation zu je einer Lifeline für jede Portinstanz dieser Komponenteninstanz **(2)**. Kommuniziert in einem MSD der alten Spezifikation eine konkrete Lifeline also nur mit einer einzigen anderen Lifeline, dann wird in der neuen Spezifikation eine entsprechende Lifeline erzeugt, die genau eine Portinstanz repräsentiert. Im MSD im Beispiel ist das für pC der Fall. Kommuniziert die ursprüngliche Lifeline jedoch mit mehr als einer anderen Lifeline, dann wird diese Lifeline im neuen MSD in je eine Lifeline pro Kommunikationspartner aufgeteilt. Jede von diesen neuen Lifelines repräsentiert dann eine andere Portinstanz derselben Komponenteninstanz. Dies ist im MSD im Beispiel für aC der Fall.

Ist eine Lifeline in der ursprünglichen MSD-Spezifikation symbolisch, dann kann sie je nach Bindung eines von mehreren Objekten der gleichen Klasse repräsentieren (siehe Abschnitt 2.1.4). Analog zur Behandlung konkreter Lifelines werden auch symbolische Lifelines in der neuen Spezifikation zu je einer Lifeline für jede Kommunikation mit anderen Lifelines. Auch diese neuen Lifelines sind symbolisch; ihr Typ ist jeweils durch diejenige Klasse definiert, die für den repräsentierten Port erstellt wurde.

Zusätzlich müssen für alle nicht an der minimalen Nachricht des MSDs beteiligten symbolischen Lifelines passende Binding Expressions erzeugt werden: War die Lifeline in der ursprünglichen Spezifikation an der minimalen Nachricht beteiligt, so empfängt im neuen MSD *eine* der abgeleiteten Lifelines die minimale Nachricht. Die übrigen abgeleiteten Lifelines werden dann per Binding Expression an die passenden übrigen Objekte gebunden, die Portinstanzen derselben Komponenteninstanz repräsentieren. Um dies zu ermöglichen, werden geeignete Assoziationen zwischen allen diesen Objekten angelegt.

Wäre die Lifeline aC im Beispiel also symbolisch, so müsste für die neue Lifeline aC→pC eine Binding Expression angelegt werden, z. B. „aC←s.pC“, wenn der Port zur Kommunikation mit pC durch die Assoziation pC von der gebundenen Lifeline aC←s aus erreichbar ist. Für symbolische Lifelines der ursprünglichen Spezifikation, die nicht an der minimalen Nachricht beteiligt sind, muss bereits jeweils eine Binding Expression existieren. Für die neuen Lifelines, die von dieser abgeleitet werden, werden diese Binding Expressions angepasst übernommen.

Um die ursprünglich definierte Reihenfolge der Nachrichten in den MSDs zu erhalten, wird zwischen je zwei Nachrichten, die in der alten Spezifikation auf derselben Lifeline aufeinanderfolgen, die aber von unterschiedlichen neuen Lifelines gesendet bzw. empfangen werden, eine heiße Bedingung `true` eingefügt. Diese Bedingung überdeckt alle Lifelines von Portinstanzen derselben Komponenteninstanz und erzwingt ihre Synchronisation. Im Beispiel ist das für die beiden Lifelines erforderlich, die für aC erstellt wurden.

Nach der Ableitung der angepassten Spezifikation wird der Synthesealgorithmus mit der neuen MSD-Spezifikation auf Portebene als Eingabe ausgeführt **(3)**. Nach der Synthese werden durch das Werkzeug „MechatronicUML-Export“ im MechatronicUML-Modell Komponenten generiert, die den Klassen der alten Spezifikation entsprechen, und Ports, die den Klassen der neuen Spezifikation entsprechen. Die generierten Controller werden direkt in äquivalente Port-RTSCs übersetzt. Die Komponenten-RTSCs entstehen durch Kombination aller Port-RTSCs der jeweiligen Komponente in parallelen Regionen eines einzigen komplexen Zustands.

Da in MechatronicUML jede Kommunikation zwischen Komponenten durch ein Echtzeitkoordinationsprotokoll beschrieben werden muss, wird für jede derartige Kommunikationsverbindung ein solches erzeugt. Durch Kopieren der Port-RTSCs wird jeweils ein identisches Rollen-RTSC erzeugt, da in MechatronicUML jeder Port eine Rolle implementieren muss. Auch hier werden aus der Strukturdefinition der MSD-Spezifikation entsprechende Komponenteninstanzen abgeleitet – ähnlich, wie es in Abschnitt 5.1.2 beschrieben wurde.

5.2 Integration in den MechatronicUML-Prozess

Basierend auf dem in der MechatronicUML-Spezifikation definierten *Entwicklungsprozess* [BDG+14b] wird hier betrachtet, wie das Syntheseverfahren im Kontext der Methode MechatronicUML sinnvoll verwendet werden kann.

In Abschnitt 5.2.1 wird zunächst der Standardablauf behandelt – für den Fall, dass manuelle Erweiterungen erforderlich sind. In Abschnitt 5.2.2 wird dann untersucht, in welchen Fällen Teile des Prozesses wiederholt oder verändert ausgeführt werden müssen. Dabei wird zunächst von einer gleichbleibenden MSD-Spezifikation ausgegangen. In Abschnitt 5.2.3 wird dann diskutiert, wie mit veränderten Anforderungen oder Annahmen – und damit einer veränderten MSD-Spezifikation – umgegangen werden kann.

5.2.1 Standardablauf

In Abschnitt 1.3 wurde bereits ein sehr abstrakter Entwicklungsprozess vorgestellt. Wie dort beschrieben wurde, müssen die Entwickler die Anforderungen zunächst mittels MSDs spezifizieren. Diese dienen als Eingabe des Syntheseverfahrens, dessen Anwendung in einem automatisierten Schritt folgt. Die erzeugten Controller werden dann bei Bedarf manuell weiter verfeinert und um zusätzliche, insbesondere plattformspezifische, Informationen ergänzt, um schließlich automatisiert Quellcode zu erzeugen. Dieser Prozess wird hier verfeinert und an MechatronicUML angepasst.

Grundsätzlich besteht bei manuellen Änderungen der Controller bzw. der abgeleiteten RTSCs die Gefahr, dass das System anschließend nicht mehr die Spezifikation erfüllt. Zudem kann es auch an das nicht mit MSDs spezifizierte Verhalten Anforderungen geben. Daher wird im Folgenden nach der Beschreibung der manuellen Weiterentwicklung diskutiert, wie sichergestellt werden kann, dass das System sowohl die MSD-Spezifikation als auch weitere Anforderungen erfüllt.

Abbildung 5.4 zeigt einen Entwicklungsprozess für verteilte mechatronische Systeme, der für die Integration der Synthese angepasst wurde. Er orientiert sich am Mecha-

tronicUML-Entwicklungsprozess [BDG+14b, GRSS11]. Die meisten der dargestellten Aktionen müssen nur bei Bedarf ausgeführt werden. Die einzelnen Schritte werden im Folgenden beschrieben.

Spezifikation und Validierung der Anforderungen und Durchführen der Synthese Zunächst spezifizieren Entwickler die Systemanforderungen und Umgebungsannahmen in einer MSD-Spezifikation und validieren diese durch Play-out [BGP13, BGH+14], also durch schrittweises Simulieren der erlaubten Nachrichtenfolgen. Diese Schritte werden durch Werkzeuge (siehe Abschnitt 6.1.1) unterstützt, sind jedoch in erster Linie manuell durchzuführen. Wenn die Entwickler bei der Simulation feststellen, dass das modellierte Verhalten nicht den tatsächlich umzusetzenden Anforderungen entspricht, sind Änderungen der MSD-Spezifikation erforderlich. Sobald die Entwickler der Ansicht sind, dass die Spezifikation die notwendigen Anforderungen und Annahmen korrekt erfasst, kann die verteilte Synthese durchgeführt werden. Dabei wird hier von einer vorherigen Adaption der MSD-Spezifikation auf Port-Ebene ausgegangen, wie sie in Abschnitt 5.1.3 beschrieben wurde.

Weiterentwicklung auf Basis des erzeugten Modells

Für eine konsistente Spezifikation resultiert die Synthese in verteilten zeitbehafteten Controllern, die die Spezifikation umsetzen. Aus ihnen und aus dem Strukturmodell der Spezifikation kann ein initiales MechatronicUML-Modell abgeleitet werden. Dieses enthält Komponenten, deren Kommunikationsverhalten bereits durch RTSCs definiert ist (siehe Abschnitt 5.1.3).

Die generierten RTSCs sind semantisch identisch zu den synthetisierten Controllern und stellen damit zunächst eine korrekte Implementierung der MSD-Spezifikation dar. Wie bereits in der Einleitung dieses Kapitels motiviert wurde, können jedoch aus verschiedenen Gründen manuelle Änderungen oder Erweiterungen erforderlich sein. Diese werden in den weiteren Prozessschritten nach der Synthese durchgeführt. Zusätzliche Schritte sind erforderlich, um sicherzustellen, dass auch die veränderten Modelle die MSD-Spezifikation einhalten.

Dekomposition der Komponentenstruktur Die initial erzeugte Komponentenstruktur entspricht der Strukturdefinition der MSD-Spezifikation. Es kann jedoch sinnvoll sein, die Struktur weiter aufzuteilen, um die Komplexität des Verhaltens einzelner Komponenten oder Ports zu reduzieren und dadurch ihre Wartbarkeit zu erhöhen. Eine weitere Motivation für eine Aufteilung kann eine geplante Allokation der Software auf unterschiedliche Hardware sein, was in MechatronicUML eine Aufteilung in mehrere Komponenten erfordert [DP14].

MechatronicUML ermöglicht die Spezifikation von *strukturierten Komponenten*, deren Verhalten nicht direkt durch RTSCs definiert wird, sondern die sämtliche Kommunikation an Subkomponenten delegieren. Da die Synthese jedoch zunächst nur das Verhalten für atomare Komponenten erzeugt, muss für eine nachträgliche Dekomposition einer atomaren Komponente in eine strukturierte Komponente das Komponenten-

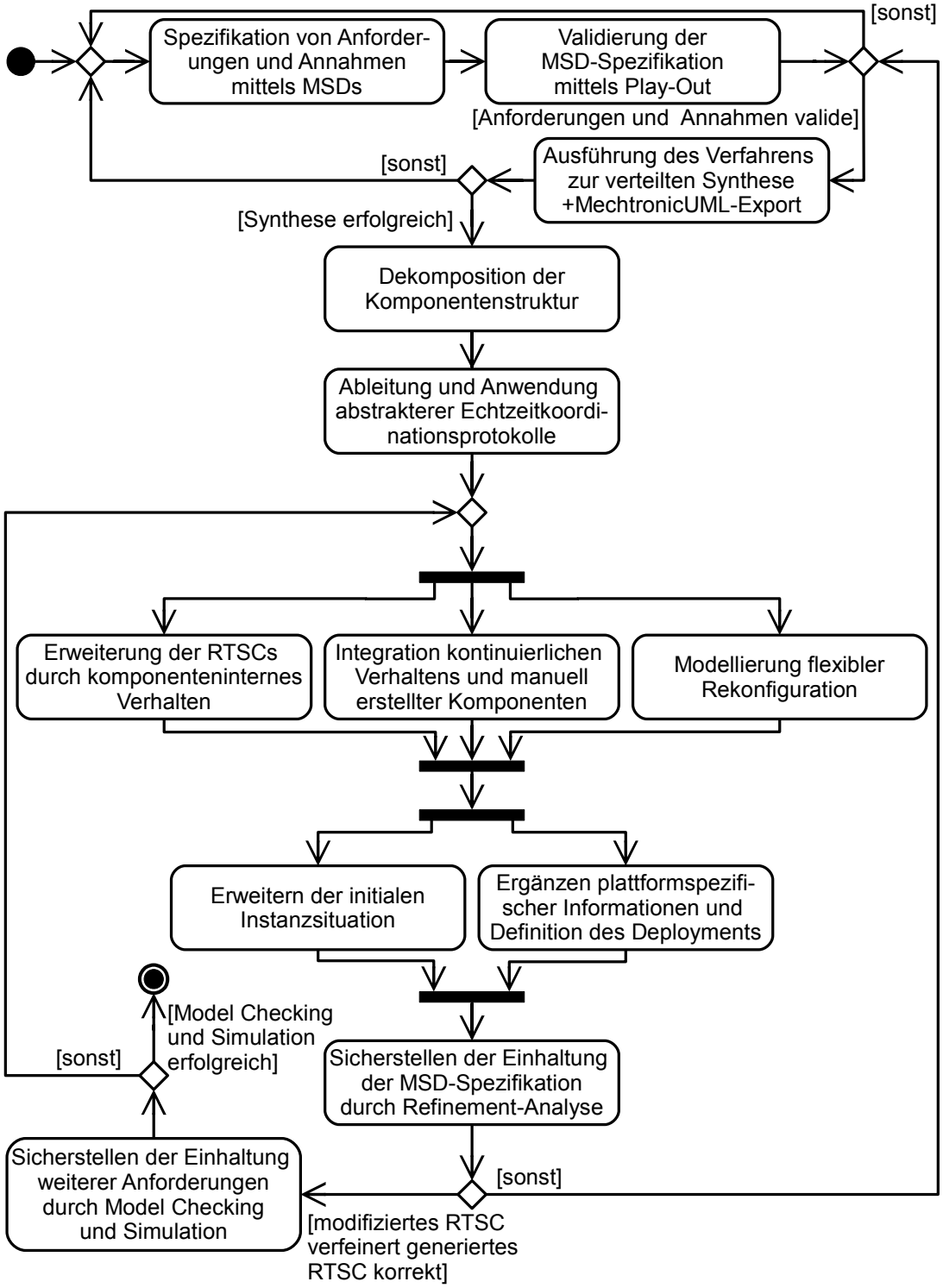


Abbildung 5.4: Entwicklungsprozess bei Verwendung des Syntheseansatzes

RTSC entsprechend auf die neuen atomaren (Sub-)Komponenten der neuen strukturierten Komponente aufgeteilt werden. Die Entwickler müssen eine solche Aufteilung so durchführen, dass das Gesamtverhalten der Komponente identisch bleibt.

Einfacher ist der umgekehrte Weg, also das Zusammenfassen mehrerer generierter atomarer Komponenten zu einer strukturierten Komponente. Hier muss nur die neue Struktur angelegt werden, in der das Portverhalten der strukturierten Komponente an die passenden Ports der Subkomponenten delegiert wird.

Neben der Aufteilung von Komponenten ist eine weitere Dekomposition der Struktur durch Aufteilung einzelner Ports in mehrere möglich. Aus dem Ergebnis der Synthese wird nach dem Schema aus Abschnitt 5.1.3 für jede Kommunikation zweier Komponenten nur jeweils ein Port in beiden beteiligten Komponenten erzeugt. Dabei kann jedoch das Verhalten zur Umsetzung mehrerer eigentlich separater Funktionen miteinander vermischt werden. Eine manuelle Aufteilung in mehrere Ports zur Kommunikation mit *derselben* Komponente – und damit auch eine Aufteilung des betreffenden Port-RTSCs – reduziert die Komplexität der einzelnen Ports. Eine solche Aufteilung muss aber auch auf Ebene der Rollen analog durchgeführt werden – und für den Port bzw. die Rolle der Gegenseite.

Ableitung und Anwendung abstrakterer Echtzeitkoordinationsprotokolle Die nach dem Schema in Abschnitt 5.1.3 erzeugten Echtzeitkoordinationsprotokolle beschreiben jeweils nur das Kommunikationsverhalten zweier Ports. Sie abstrahieren also nicht von der konkreten Anwendung, was der Grundidee dieser Protokolle widerspricht. Als manueller Prozessschritt sollten daher die automatisch erzeugten Echtzeitkoordinationsprotokolle zu abstrakteren und allgemeiner anwendbaren zusammengefasst werden.

Alternativ zum Anpassen bzw. Zusammenfassen der automatisch generierten Echtzeitkoordinationsprotokolle können diese verworfen und stattdessen bereits existierende Protokolle angewendet werden. Diese müssen dazu aber zunächst überhaupt verfügbar sein. Für die Wiederverwendung von Echtzeitkoordinations*protokollen* wurde in MechatronicUML das Konzept der Echtzeitkoordinations*muster* eingeführt, die von ersteren abstrahieren und die in Musterkatalogen gesammelt werden können [DHT12, DBHT12, PDM+14].

Sofern die geänderten oder übernommenen Echtzeitkoordinationsprotokolle ohne Änderung der automatisch erzeugten Port-RTSCs angewendet werden können, ist die Einhaltung der MSD-Spezifikation weiterhin garantiert. Müssen die Port-RTSCs jedoch den für das übernommene Protokoll definierten Rollen-RTSCs angepasst werden, dann entspricht das Systemverhalten nicht mehr zwangsläufig dem synthetisierten Verhalten. Die Einhaltung der MSD-Spezifikation muss dann separat durch andere Verfahren gezeigt werden. Dieses Problem wird im nächsten Abschnitt behandelt.

Erweiterung der RTSCs durch komponenteninternes Verhalten Die automatisch erzeugten Komponenten enthalten zunächst nur das Verhalten, das für eine korrekte diskrete Kommunikation der Komponenten erforderlich ist. Neben der direkten Kommunikation der Komponenten miteinander gehört dazu auch die dafür möglicherweise erforderliche Synchronisation der Ports innerhalb der Komponenten.

Berechnungen, die auf Grundlage von Nachrichtenparametern oder Speicherinhalten erfolgen, können in MSDs jedoch nur sehr eingeschränkt spezifiziert werden. Komplexere Operationen mit beispielsweise Zeichenketten, Zahlen oder Arrays müssen daher manuell in den erzeugten RTSCs ergänzt werden. Die MechatronicUML ermöglicht es, solche Berechnungen als *Seiteneffekte* von Transitionen und als *Aktionen* in Zuständen zu spezifizieren, wobei auf interne Variablen des jeweiligen RTSCs zugegriffen werden kann [BDG+14b].

Integration kontinuierlichen Verhaltens und manuell erstellter Komponenten Kontinuierliches Verhalten muss in mechatronischen Systemen beispielsweise zur Ansteuerung von Aktoren, dem Auslesen von Sensorwerten und der Kommunikation mit kontinuierlichen Reglern berücksichtigt werden. In MechatronicUML kann dieses kontinuierliche Verhalten zwar nicht direkt spezifiziert werden, es kann aber nach Spezifikation mit anderen Werkzeugen mittels spezieller *kontinuierlicher Komponenten* integriert werden. In *hybriden strukturierten Komponenten* können sowohl kontinuierliche als auch diskrete Komponenten eingebettet werden und diese Verhaltensbestandteile somit kombiniert werden.

Da kontinuierliches Verhalten nicht mit dem in dieser Arbeit vorgestellten Verfahren synthetisiert werden kann, ist dessen Modellierung und die Integration in das erzeugte Modell ein komplett manueller Schritt. Dabei müssen die Entwickler jedoch sicherstellen, dass RTSCs, die auf kontinuierliches Verhalten reagieren, kein gegenüber den erzeugten Controllern verändertes diskretes Kommunikationsverhalten beschreiben dürfen.

Unterstützung flexibler Rekonfiguration In vielen mechatronischen Systemen ist es für Subsysteme erforderlich, dynamisch auf Änderungen in der Umgebung oder in anderen Subsystemen zu reagieren. Das Subsystem muss dann für unterschiedliche Situationen oft ein sehr unterschiedliches Verhalten definieren. In MSD-Spezifikationen ist es mittels symbolischer Lifelines eingeschränkt möglich, solches situationsabhängiges Verhalten zu modellieren: Das von einem Objekt geforderte Verhalten kann davon abhängen, welche von mehreren symbolischen Lifelines (siehe Abschnitt 2.1.4) eines MSDs an dieses Objekt gebunden ist.

Die Synthese erzeugt die Controller so, dass das Verhalten der Objekte in jeder Situation – also auch für alle symbolischen Lifelines, an die sie gebunden sein können – korrekt ist. Das für die verschiedenen Situationen relevante Verhalten ist dann jedoch nicht voneinander und ggf. von gemeinsamen Verhaltensbestandteilen getrennt. Eine Trennung dieser Verhaltensbestandteile könnte aber dazu beitragen, die Komplexität der Controller zu reduzieren und das Verhalten in den einzelnen Situationen besser zu kapseln. Eine weitere Motivation für eine solche Trennung ist der potentiell reduzierte Ressourcenbedarf dadurch, dass nur jeweils ein Teil des Gesamtverhaltens aktiv sein muss.

MechatronicUML unterstützt eine flexible Rekonfiguration der komponenteninternen Struktur zur Laufzeit [HB13, EHH+13], durch die auch die verwendeten Verhaltensmodelle ausgetauscht werden. Das Rekonfigurationsverhalten, also das Verhalten, welches die Entscheidung zur Strukturänderung trifft und diese durchführt, wird sepa-

rat vom übrigen Verhalten der Komponenten definiert. Um eine automatisch erzeugte Komponente nachträglich so zu verändern, dass diese flexible Rekonfiguration unterstützt, muss das Verhalten des Komponenten-RTSCs daher manuell in das für die jeweilige Konfiguration spezifische Verhalten und das Rekonfigurationsverhalten aufgeteilt werden. Auch die komponenteninterne Struktur muss dafür angepasst werden.

Einfacher als eine nachträgliche Aufteilung des Verhaltens kann es sein, das in unterschiedlichen Situationen geforderte Verhalten auf Ebene der MSD-Spezifikation zunächst *separat* zu definieren. Ein Subsystem kann beispielsweise in der MSD-Spezifikation durch mehrere Objekte modelliert werden, die das Subsystem jeweils nur in *einer* Situation repräsentieren. Anders als bei der Modellierung durch ein Objekt und symbolische Lifelines, können diese Objekte dann nicht mehr ihre „Rollen“ in den einzelnen Szenarien tauschen.

Die durch die Synthese für eine Spezifikation erzeugten Controller bzw. die abgeleiteten Komponenten beschreiben dann jeweils nur das Verhalten des Subsystems in einer dieser Situationen. Sie können dann durch den Entwickler zu einer einzigen Komponente kombiniert werden, die zwischen den unterschiedlichen Verhaltensbestandteilen per dynamischer Rekonfiguration umschaltet. Ein Nachteil hierbei ist jedoch, dass das Rekonfigurationsverhalten selbst vollständig manuell spezifiziert werden muss.

Eine explizite Modellierung dynamischer Rekonfiguration auf Ebene der Spezifikation kann möglicherweise durch eine Kombination von MSDs und Graphtransformationssystemen [WGT14] erzielt werden. Eine Anwendung des hier vorgestellten Syntheseansatzes auf diese neue Spezifikationstechnik sprengt jedoch den Rahmen dieser Arbeit, da für diese Kombination neue Konzepte entwickelt werden müssten.

Weitere Schritte Beim Ableiten eines MechatronicUML-Modells aus dem dem Syntheseergebnis wird auch ein Modell einer initialen Instanzsituation erzeugt; dieses entspricht aber zunächst nur der in der MSD-Spezifikation im Kompositionsstrukturdiagramm definierten Situation. Es muss beispielsweise um Instanzen von gegebenenfalls vorhandenen kontinuierlichen Komponenten erweitert werden.

Nach Fertigstellung des Design-Modells müssen diesem plattformspezifische Informationen hinzugefügt werden, um schließlich eine automatische Codegenerierung zu ermöglichen. Dazu gehört beispielsweise das Definieren eines *Deployments*, also der Zuordnung von Komponenteninstanzen zur Hardware, die diese ausführt. Für eine genauere Beschreibung dieser Schritte wird auf Dann und Pohlmann [DP14] verwiesen.

Sicherstellen der Korrektheit

Um ein korrektes Systemverhalten sicherzustellen, wurden zur Verwendung im Rahmen der MechatronicUML-Methode in früheren Arbeiten (teil-)automatische Verfahren entwickelt, mit denen MechatronicUML-Modelle auf Korrektheit überprüft werden können [DGB+14]. Dies sind Verfahren zum Model Checking, zur Verfeinerungsanalyse und zur Simulation. Mit diesen kann einerseits überprüft werden, ob die MSD-Spezifikation nach manuellen Änderungen an den automatisch erzeugten RTSCs noch eingehalten wird. Andererseits kann die Einhaltung zusätzlicher Anforderungen überprüft

werden, die auf Ebene der MSD-Spezifikation nicht sinnvoll definiert werden können. Dazu zählen beispielsweise Anforderungen an kontinuierliches Verhalten.

Sicherstellen der Einhaltung der MSD-Spezifikation Wenn das verteilte System das Kommunikationsverhalten aufweist, das durch die per Synthese erzeugten Controller definiert wird, dann erfüllt es die MSD-Spezifikation. Jedes Subsystem muss also die durch seinen Controller vorgegebenen Nachrichten entsprechend der in Abschnitt 2.2 definierten Controller-Semantik senden, also als Antwort auf die richtigen Eingaben und unter Einhaltung der erzeugten Time Guards und Invarianten. Manuelle Änderungen an den automatisch erzeugten Modellen bergen jedoch die Gefahr, dass sich dieses Kommunikationsverhalten so ändert, dass es nicht mehr dem Verhalten der erzeugten Controller entspricht – und es somit potentiell die MSD-Spezifikation verletzt.

Systemverhalten, das nicht die in den MSDs definierten Anforderungen betrifft, darf beliebig verändert werden. Es kann jedoch subtile Abhängigkeiten zwischen den Verhaltensbestandteilen geben, die auch dann zu einer Verletzung der MSD-Spezifikation führen können, wenn das geänderte Verhalten nicht direkt die Kommunikation betrifft. Daher sollte das manuell erweiterte Verhalten durch Kapselung möglichst vollständig vom generierten Verhalten getrennt werden, beispielsweise durch Definition in einer eigenen Komponente. Wenn aber direkte Abhängigkeiten zwischen dem generierten und dem manuell erzeugten Verhalten für ein korrektes Systemverhalten erforderlich sind, dann ist eine solche Trennung kaum möglich. In diesen Fällen kann aber durch ein spezielles automatisches Verfahren geprüft werden, ob das Kommunikationsverhalten durch die Änderungen unverändert geblieben ist.

In MechatronicUML sind Verfahren zur *Verfeinerungsanalyse* integriert, die eine automatische Prüfung ermöglichen, ob das durch ein Verhaltensmodell definierte externe Kommunikationsverhalten identisch zu dem eines anderen Modells ist [HBDS15, BHSH13]. Diese Verfahren werden in MechatronicUML zur Überprüfung verwendet, ob die Port-RTSCs die Rollen-RTSCs korrekt verfeinern. Wenn das der Fall ist, dann kann aus einem korrekten Verhalten der Rollen auf ein korrektes Verhalten der Ports geschlossen werden.

Mittels dieser Verfahren kann aber allgemein für ein manuell modifiziertes Verhaltensmodell überprüft werden, ob es in dieser Hinsicht immer noch dem durch die Synthese erzeugten Modell entspricht. Dazu muss das automatisch erzeugte Komponenten-RTSC vor der manuellen Veränderung separat gespeichert werden. Anschließend kann dann mittels der Verfeinerungsanalyse geprüft werden, ob das externe Kommunikationsverhalten der Komponente für beide Varianten des Komponenten-RTSCs identisch ist. In diesem Fall wird die MSD-Spezifikation auch durch das veränderte System erfüllt.

Um ein unverändertes Kommunikationsverhalten der RTSCs nachzuweisen, muss auf Vorliegen einer *Timed Bisimulation* [WL97] geprüft werden. Diese Verfeinerungsbeziehung garantiert nicht nur, dass dieselben Sequenzen von Ein- und Ausgaben auftreten können, sondern auch, dass das Senden bzw. Empfangen jeweils in denselben Zeitintervallen stattfindet. Durch den Nachweis, dass die manuell geänderten Komponenten-RTSCs immer noch den generierten Controllern (bzw. den abgeleiteten RTSCs) entsprechen, ist garantiert, dass diese weiterhin die MSD-Spezifikation einhalten.

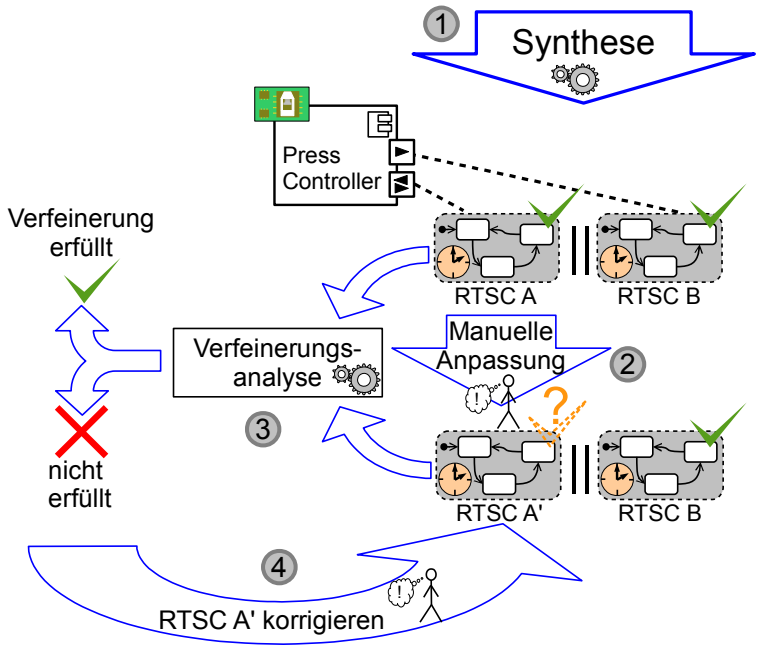


Abbildung 5.5: Anwendung der Verfeinerungsanalyse zum Nachweis der Korrektheit von manuell modifiziertem Verhalten

Abbildung 5.5 zeigt am Beispiel eines manuell modifizierten Port-RTSCs des Steuergeräts der Presse, wie das Verfahren zur Verfeinerungsanalyse verwendet werden kann, um nachzuweisen, dass auch das veränderte Modell noch die MSD-Spezifikation einhält: Nach der Synthese **(1)** wird vom manuell modifizierten **(2)** RTSC vor dieser Modifikation eine Kopie angelegt. Anschließend wird mittels der Verfeinerungsanalyse **(3)** überprüft, ob das gesamte modifizierte Komponenten-RTSC (also einschließlich des unmodifizierten RTSCs des anderen Ports) eine Timed Bisimulation zum gesamten unmodifizierten Komponenten-RTSC einhält. Ist dies der Fall, so ist auch das modifizierte Verhalten der Komponente korrekt. Andernfalls **(4)** muss das modifizierte RTSC solange manuell korrigiert werden, bis die Verfeinerungsanalyse erfolgreich ist.

Sicherstellen der Einhaltung weiterer Anforderungen Sofern neben den in der MSD-Spezifikation definierten Anforderungen an die nachrichtenbasierte Kommunikation *zusätzliche* Anforderungen berücksichtigt werden sollen, so müssen diese separat verifiziert werden. Beispielsweise können Anforderungen hinzukommen, die sich auf konkrete RTSC-Zustände beziehen. Diese existieren auf Ebene der MSD-Spezifikation zunächst noch nicht und können daher nicht in ihr referenziert werden. Indirekt können zustandsbezogene Anforderungen in MSDs durch Bezug auf die Nachrichten definiert werden, durch die der Zustand betreten und verlassen wird. Dies kann jedoch sehr umständlich sein und widerspricht dem nachrichtenbasierten Paradigma der MSDs. Ein direkter Bezug auf die Controllerzustände (bzw. auf Zustände der abgeleiteten RTSCs) andererseits

ist jedoch in MSDs nicht möglich, da diese Zustände erst durch die Synthese erzeugt werden.

Im SceBaSy-Ansatz [Gie03, GB04, GT05] ist eine Festlegung von Zuständen mit konkreter Benennung in einer anderen Form von Sequenzdiagrammen möglich, aus denen dann RTSCs abgeleitet werden können. Dazu müssen die Anforderungen jedoch auf einer im Vergleich zu MSDs konkreteren Ebene spezifiziert werden. Zudem ist dieser Ansatz nicht mit der aktuellen Werkzeugunterstützung für MechatronicUML kompatibel.

Um Eigenschaften, die sich bereits auf konkrete RTSC-Zustände beziehen, automatisch verifizieren zu können, wurde in MechatronicUML ein zeitbehaftetes *Model Checking* mittels des Werkzeugs Uppaal[1] [BDL04] integriert [BFG+03, DGB+14]. Die zu prüfenden Eigenschaften müssen dazu in einer zeitbehafteten Temporallogik definiert werden, die auf TCLT [ACD90], einer zeitbehafteten Erweiterung der CTL [BK07], basiert. Diese Anforderungen können entweder auf Ebene der Echtzeitkoordinationsprotokolle – also als Anforderungen an das Rollenverhalten – oder als Anforderungen an das konkrete Komponentenverhalten spezifiziert werden.

Da die kontinuierlichen Anteile des Systemverhaltens nicht durch das Syntheseverfahren abgedeckt werden und auch dem in MechatronicUML umgesetzten Model Checking nicht zugänglich sind, kann die Korrektheit dieser Verhaltensbestandteile nicht mit der Werkzeugunterstützung für MechatronicUML nachgewiesen werden. Die korrekte Entwicklung dieser Verhaltensbestandteile wird im MechatronicUML-Prozess jedoch ohnehin nicht als Aufgabe der Softwaretechnik gesehen. Dieses Verhalten kann jedoch mit externen Werkzeugen, z. B. MATLAB/Simulink, entwickelt und dann in Form spezieller *kontinuierlicher Komponenten* in MechatronicUML eingebunden werden. Das Gesamtverhalten kann dann simuliert werden, beispielsweise ebenfalls durch Verwendung von MATLAB/Simulink [HRS13]. Dies ermöglicht eine manuelle Fehlersuche durch Testen: Die Entwickler des Systems vergleichen dazu das simulierte Systemverhalten mit dem erwarteten für einzelne konkrete Ausführungen [DGB+14].

5.2.2 Zusätzliche Iterationen und Abweichungen vom Standardfall

Beim beschriebenen Standardfall des Entwicklungsprozesses wurde davon ausgegangen, dass alle Entwicklungsschritte mit idealem Ergebnis abgeschlossen werden. Dies ist jedoch bei einer realen Systementwicklung oft nicht der Fall. Es kann dann erforderlich sein, dass bestimmte Schritte des Prozesses wiederholt werden oder zusätzliche bzw. alternative Schritte durchgeführt werden.

Fehlschlagen der Synthese Im oben dargestellten Standardfall des Prozesses wird davon ausgegangen, dass das Ergebnis der Anforderungs-Modellierung eine konsistente MSD-Spezifikation ist, für die eine verteilte Umsetzung möglich ist. Wurden jedoch widersprüchliche Anforderungen modelliert oder sind diese nicht verteilt realisierbar, dann terminiert der Synthesealgorithmus ohne Lösung und meldet dies dem Benutzer. Der Entwickler kann dann anhand des bei der Synthese erzeugten Spezifikationsgraphen und

[1] `http://www.uppaal.org/`

der erfolglos erzeugten Controllersystem-Kandidaten im Kandidatengraph untersuchen, worin der Grund für den Fehlschlag besteht. Dabei kann auch die Schritt-für-Schritt-Simulation des Systemverhaltens durch Play-out [BGP13, BGH+14] helfen. Anhand der dabei gewonnenen Erkenntnisse können die Entwickler dann die Modellierung der Anforderungen und Annahmen in der MSD-Spezifikation anpassen. Anschließend kann die Synthese erneut ausgeführt werden. Diese Schritte sind so lange zu wiederholen, bis die Synthese erfolgreich ein Controllersystem erzeugen konnte.

Fehlschlagen der Verfeinerungsanalyse Wie bereits beschrieben wurde, können manuelle Änderungen der durch Synthese erzeugten RTSCs zu einem abweichenden Kommunikationsverhalten führen. Wenn dies der Fall ist, dann wird bei der Verfeinerungsanalyse festgestellt, dass eines der manuell geänderten RTSCs keine korrekte Verfeinerung eines anderen RTSCs ist, das von einem automatisch erzeugten Controller abgeleitet wurde. Die Einhaltung der MSD-Spezifikation ist dann nicht mehr garantiert. Der Entwickler kann daraufhin seine Modifikationen rückgängig machen bzw. so abändern, dass das Kommunikationsverhalten identisch ist.

Wenn eine Verfeinerungsanalyse fehlschlägt, dann muss dies jedoch nicht unbedingt bedeuten, dass das geänderte Verhalten inkorrekt ist. Es entspricht dann zwar nicht mehr dem erzeugten Controller, kann aber dennoch die Spezifikation erfüllen. Dies ist möglich, wenn die Spezifikation alternative Implementierungen zulässt. Das geänderte Modell entspricht dann einem anderen gültigen Controller, nur eben nicht dem durch die Synthese erzeugten. Die korrekte Verfeinerung des generierten RTSCs durch das manuell veränderte ist also eine hinreichende Bedingung für die Einhaltung der MSD-Spezifikation, jedoch keine notwendige. Dies ist immer dann relevant, wenn mehrere korrekte Implementierungsvarianten für die gegebene MSD-Spezifikation existieren. In diesem Fall hilft die Verfeinerungsanalyse nicht weiter.

Ohne automatisches Verfahren kann es schwierig sein, die Korrektheit eines modifizierten Controllers gemäß der MSD-Spezifikation festzustellen: Durch Änderungen an einem der Controller können konsistente Änderungen an anderen Controllern erforderlich sein, um ein korrektes Gesamtverhalten zu erzielen. Für die mit MSDs verwandten LSCs existieren Verfahren zum Model Checking, mit denen die Konformität eines Systems zu einer gegebenen LSC-Spezifikation überprüft werden kann [SD05a, KTWW06]. Diese Verfahren sind jedoch nicht direkt auf MSDs und RTSCs übertragbar. Basierend auf dem hier vorgestellten Syntheseverfahren könnte ein Ansatz zum Model Checking von RTSCs anhand von MSD-Spezifikationen entwickelt werden. Dies würde jedoch den Rahmen dieser Arbeit sprengen und verbleibt daher als mögliche zukünftige Arbeit.

Diagnose fehlerhaften Verhaltens durch Model Checking oder Simulation Wird durch das Model Checking die Verletzung einer zu verifizierenden Eigenschaft festgestellt, so müssen die RTSCs entsprechend manuell korrigiert werden bis das Model Checking alle Eigenschaften als erfüllt meldet. Anschließend muss – wie bei anderen manuellen Änderungen – durch eine (erneute) Verfeinerungsanalyse nachgewiesen werden, dass das Kommunikationsverhalten dem der synthetisierten Controller entspricht. Bei Fehlschlag treten die oben diskutierten Konsequenzen auf.

Wird durch die Simulation ein fehlerhaftes kontinuierliches Verhalten festgestellt, so muss dieses ebenfalls korrigiert werden. Sofern das jedoch keine Auswirkungen auf das diskrete Systemverhalten hat, betrifft diese Korrektur nicht die Einhaltung der MSD-Spezifikation.

5.2.3 Behandlung von Änderungen der MSD-Spezifikation

Bei der bisherigen Beschreibung der Weiterentwicklung von MechatronicUML-Modellen nach deren Synthese wurde von einer unveränderlichen MSD-Spezifikation ausgegangen. Bei der realen Entwicklung mechatronischer Systeme sind jedoch oft nachträglich Änderungen oder Erweiterungen der Anforderungen oder der Umgebungsannahmen erforderlich. Dies kann einerseits daran liegen, dass die Spezifikation (trotz Validierung durch Play-out-Simulation) zunächst unvollständig oder fehlerhaft war. Es können sich andererseits aber auch zuvor nicht absehbare Änderungen in den Anforderungen oder der Einsatzumgebung des Systems ergeben. Die nachfolgend diskutierten Probleme betreffen nicht nur aus Controllern abgeleitete RTSCs, sondern Controller im Allgemeinen.

Anpassung der Implementierung Im Spezialfall, dass die Änderungen der MSD-Spezifikation ausschließlich in der Relaxierung von Anforderungen oder der Verstärkung von Umgebungsannahmen bestehen, erfüllen die vor diesen Änderungen synthetisierten Controller weiterhin die MSD-Spezifikation. Eine Relaxierung bedeutet dabei, dass zusätzliches Verhalten des Systems oder seiner Umgebung zugelassen wird, eine Verstärkung bedeutet das Gegenteil.

Auch wenn die Änderungen an der Spezifikation Anforderungen verstärken oder Umgebungsannahmen relaxieren kann es sein, dass die erzeugten Controller weiterhin eine gültige Implementierung der MSD-Spezifikation sind: Die strikteren Anforderungen betreffen möglicherweise nur andere mögliche Controller-Varianten, aber nicht die tatsächlich erzeugten. Ebenso sind schwächere Umgebungsannahmen unschädlich, solange die erzeugten Controller nicht auf die ursprünglichen strikteren Annahmen angewiesen sind. Für eine automatisierte Überprüfung, ob die Controller auch die geänderte MSD-Spezifikation noch einhalten, wäre jedoch ein MSD-basiertes Model Checking der Controller erforderlich. Da ein solches Verfahren nicht existiert und eine Neuentwicklung den Rahmen dieser Arbeit sprengen würde, bleibt diese Überprüfung ein manueller Schritt.

Erfüllen die erzeugten Controller (bzw. die daraus abgeleiteten RTSCs) die geänderte MSD-Spezifikation nicht mehr, so können die Entwickler die Controller manuell entsprechend dieser Änderungen anpassen, um sie wieder in eine korrekte Implementierung zu überführen. Je nach Umfang der erforderlichen Anpassungen kann das jedoch zu aufwendig oder aufgrund zu hoher Komplexität zu fehleranfällig sein. Alternativ können durch erneute Ausführung der Synthese neue Controller generiert werden, die der geänderten Spezifikation entsprechen. Dies ist jedoch nur dann sinnvoll, wenn seit der letzten Synthese keine manuellen Änderungen an den erzeugten Controllern durchgeführt wurden – oder nur solche, die sich mit vertretbarem Aufwand auf die neu generierten Controller übertragen lassen.

Um Aufwand zu sparen und Fehler zu vermeiden, wäre eine Erweiterung des in dieser Arbeit entwickelten Syntheseverfahrens sinnvoll, die manuelle Änderungen an den

Controllern bei weiteren Ausführungen der Synthese berücksichtigt bzw. in die neuen Controller integriert. Die Integration von beliebigen manuellen Änderungen ist jedoch kein triviales Problem und auch nicht immer möglich, insbesondere wenn diese manuellen Änderungen zwar der alten MSD-Spezifikation nach erlaubt waren, in der neuen jedoch nicht mehr erlaubt sind. In dieser Arbeit wurde das Problem daher nicht weiter behandelt; es verbleibt also als mögliche weiterführende Arbeit.

Ersetzen der Implementierung zur Laufzeit Ist ein System, das der alten MSD-Spezifikation genügt, bereits im Einsatz, so müssen die alten Controller via Software-Update durch neue Controller ersetzt werden. Dies ist jedoch normalerweise nicht möglich, während das System noch in Betrieb ist. Andererseits ist es jedoch gerade bei verteilten Systemen nicht immer praktikabel, alle Subsysteme für ein Update zeitweise auszuschalten. In bestimmten Zuständen eines alten Controllers kann aber ein sicheres Umschalten in einen entsprechenden Zustand des neuen Controllers möglich sein. Von Panzica La Manna et al. [PGGB13] wurde untersucht, welche Bedingungen ein Zustand erfüllen muss, um ein solches Umschalten zu ermöglichen.

In einer anderen Arbeit von Greenyer et al. [GPBG13] wurde ein Syntheseansatz entwickelt, der für zwei Versionen einer Spezifikation einen dynamisch aktualisierbaren Controller erzeugt, welcher vollständige Controller-Implementierungen für die alte und für die neue Version der Spezifikation beinhaltet. Dieser Controller ersetzt dann den alten Controller des zu aktualisierenden Systems, wobei der Zustand des dynamisch aktualisierbaren Controllers dem aktuellen Zustand des alten Controllers entspricht. Der dynamisch aktualisierbare Controller wechselt dann im nächsten erreichten Zustand, in dem ein Update möglich ist, über eine „update-Transition“ in den entsprechenden Zustand des neuen Controllers.

Der existierende Ansatz für dynamisch aktualisierbare Controller befasst sich jedoch zunächst nur mit Systemen, die durch einen einzigen Controller gesteuert werden, die also nicht verteilt realisiert sind. Eine Erweiterung dieses Ansatzes für verteilte Controller, basierend auf dem in dieser Arbeit entwickelten verteilten Syntheseansatz, wäre eine mögliche zukünftige Arbeit.

Ein zusätzliches Problem beim Aktualisieren von verteilten Controllern ist, dass die einzelnen Subsysteme möglicherweise nicht unabhängig voneinander aktualisiert werden dürfen: Miteinander kommunizierende Controller, die jeweils einen anderen Versionsstand der MSD-Spezifikation umsetzen, können zueinander inkompatibel sein und dadurch zu Nachrichtensequenzen führen, die nach keiner der beiden Versionen der Spezifikation korrekt sind. Ein Lösungsansatz wäre, das Umschalten der dynamisch aktualisierbaren Controller in die neue Implementierung erst dann durchzuführen, wenn *alle* Subsysteme bereits über einen dynamisch aktualisierbaren Controller verfügen. Zudem muss das Umschalten dann gleichzeitig erfolgen, was wiederum nur dann funktionieren kann, wenn sich *alle* Subsysteme in einem aktualisierbaren Zustand befinden.

5.3 Zusammenfassung

In diesem Kapitel wurde am Beispiel von MechatronicUML untersucht, wie das in den Kapiteln 3 und 4 beschriebene Syntheseverfahren im Kontext einer existierenden Entwicklungsmethode für mechatronische Systeme verwendet werden kann.

Eine Einbettung in eine existierende Entwicklungsmethode ist einerseits erforderlich, weil durch die Synthese einige Verhaltensbestandteile – beispielsweise kontinuierliches Verhalten – nicht oder nur eingeschränkt erzeugt werden können. Andererseits kann es durch den hohen Berechnungsaufwand der verteilten Synthese erforderlich sein, diese nur für Teile eines Systems einzusetzen, oder auf einer hohen Abstraktionsebene, die das Hinzufügen von weiteren Details erfordert. In beiden Fällen sind manuelle Anpassungen durch die Entwickler erforderlich. Zudem unterstützt die Methode auch weitere Schritte auf dem Weg zur Implementierung, die nicht durch das Syntheseverfahren abgedeckt werden. Dies sind beispielsweise das Ergänzen plattformspezifischer Informationen im Modell und schließlich die Codegenerierung.

Die Integration des Syntheseverfahrens in MechatronicUML kann auf verschiedene Weisen erfolgen, die sich einerseits im Aufwand, andererseits in unterschiedlichen Möglichkeiten zur manuellen Weiterentwicklung der erzeugten Modelle unterscheiden. Üblicherweise bezieht sich eine MSD-Spezifikation auf die Kommunikation zwischen den Subsystemen eines verteilten Systems, die jeweils als Objekte modelliert werden. Eine Anwendung der Synthese auf eine solche MSD-Spezifikation erzeugt Controller, die sich nur eingeschränkt in MechatronicUML weiterentwickeln lassen. Bei geeigneter Anpassung der MSD-Spezifikation entsprechen die erzeugten Controller besser den MechatronicUML-spezifischen Vorgaben. Daher wurde in diesem Kapitel eine systematische Anpassung der MSD-Spezifikation vor der Synthese beschrieben.

In jedem Fall entsprechen die erzeugten Controller immer den Real-Time Statecharts (RTSCs) in MechatronicUML. Daher müssen die einzelnen Controller auf RTSCs abgebildet werden. Diese Abbildung ist konzeptionell sehr einfach, da zeitbehaftete Controller im Wesentlichen den Timed Automata entsprechen, während RTSCs, als Erweiterung der Timed Automata, ebenfalls alle Arten von Modellelementen unterstützen, die diese definieren.

Um das Syntheseverfahren im Rahmen der Entwicklungsmethode MechatronicUML verwendbar zu machen, muss auch der MechatronicUML-Entwicklungsprozess entsprechend angepasst werden, da die Weiterentwicklung der Modelle in diesem Prozess erfolgt. Dabei sind Anpassungen des Prozesses insbesondere notwendig, um auf manuelle Änderungen am MechatronicUML-Modell oder an der MSD-Spezifikation zu reagieren: Es muss sichergestellt sein, dass das Modell weiterhin eine korrekte Implementierung der Spezifikation bleibt – oder in eine solche überführt wird.

Nachdem dieses Kapitel den konzeptionellen Teil der Arbeit abgeschlossen hat, behandelt Kapitel 6 die Implementierung und Evaluierung der beschriebenen Konzepte.

Kapitel 6

Implementierung und Evaluierung

Dieses Kapitel stellt die Implementierung der in den Kapiteln 3 und 4 beschriebenen Konzepte und deren Evaluierung anhand von Beispielmodellen vor. Zu Kapitel 5 wurde eine Transformation von Syntheseergebnissen zu MechatronicUML-Modellen implementiert, die größtenteils dort beschrieben wird, zu der hier aber weitere Details behandelt werden.

Das Kapitel ist wie folgt unterteilt: Abschnitt 6.1 beschreibt die im Rahmen dieser Arbeit durchgeführten Implementierungen. Dazu gehört insbesondere die verteilte Synthese selbst mit den für diese vorgestellten Erweiterungen. Abschnitt 6.2 diskutiert die Evaluierung anhand der Implementierung und Abschnitt 6.3 schließlich fasst dieses Kapitel zusammen.

6.1 Implementierung

Das in dieser Arbeit beschriebene Verfahren für die verteilte Synthese wurde auf Basis der Werkzeugumgebung ScenarioTools[1] [BGP13, BGH+14] umgesetzt. Gleichzeitig kann das Syntheseverfahren selbst als neuer Bestandteil von ScenarioTools gesehen werden. Im Rahmen dieser Arbeit wurde neben der Entwicklung des eigentlichen Verfahrens aber auch der Kern von ScenarioTools weiterentwickelt und insbesondere um Echtzeitkonzepte erweitert.

ScenarioTools wurde in der ursprünglichen Version ab 2009 entwickelt [Gre11] und ab 2012 unter Mitwirkung des Autors dieser Arbeit neu implementiert [BGP13, BGH+14]. Sowohl ScenarioTools als auch die Implementierung dieser Arbeit bestehen hauptsächlich aus Plug-ins für die Entwicklungsumgebung Eclipse[2]. Die meisten dieser Plug-ins verwenden zudem das Eclipse Modeling Framework (EMF)[3], das den Austausch von Modellen zwischen den Werkzeugen erleichtert.

ScenarioTools ermöglicht das Modellieren von Spezifikationen auf Basis von Modal Sequence Diagrams (MSDs) (siehe Abschnitt 2.1) und eine Schritt-für-Schritt-Simulation des spezifizierten Verhaltens. Die Implementierung dieser Arbeit basiert auf der *MSD-Runtime*, einer Laufzeitumgebung für MSDs, die den Kern von ScenarioTools darstellt [BGP13]. Dadurch muss das Syntheseverfahren die MSDs nicht direkt analysieren, sondern es kann die MSD-Runtime zur schrittweisen Erzeugung des Spezifikationsgraphen (SG) (siehe Abschnitt 3.2.1) nutzen.

[1] `http://www.scenariotools.org`
[2] `http://www.eclipse.org`
[3] `http://www.eclipse.org/emf`

Die Verwendung der MSD-Runtime durch die Synthese hat den Vorteil, dass letztere ohne Mehraufwand exakt dieselbe Variante der Semantik von MSDs umsetzt, wie die Play-out-Simulation in ScenarioTools. In früheren Versionen von ScenarioTools [Gre11] war das noch nicht der Fall und konnte dazu führen, dass Entwickler in bestimmten Fällen abhängig von den verwendeten Modellelementen nur entweder die Synthese oder die Simulation nutzen konnten. Insbesondere unterstützte nur die Synthese Echtzeitverhalten und nur die Simulation symbolische Lifelines.

In Abschnitt 6.1.1 wird erläutert, wie die in dieser Arbeit neu entwickelte Software zusammen mit bestehenden Werkzeugen im Entwicklungsprozess verwendet werden kann. Anschließend gibt Abschnitt 6.1.2 einen Überblick über die Architektur der entwickelten Software und über die Abhängigkeiten zu anderen Werkzeugen.

6.1.1 Verwendung der Werkzeugunterstützung im Entwicklungsprozess

Hauptbestandteil der Implementierung ist ein Werkzeug zur Durchführung der verteilten Synthese, welches die Konzepte aus Kapitel 3 und zusätzliche Konzepte aus Kapitel 4 umsetzt. Dieses Werkzeug automatisiert die sonst manuell erforderliche Modellierung von spezifikationsgemäßer Kommunikationssoftware. Die dazu erforderliche Spezifikation muss zuvor mittels MSDs formal definiert werden. Weitere manuelle oder automatische Schritte der Softwareentwicklung können nach dem in Abschnitt 5.2 vorgeschlagenen Prozess im Rahmen der Entwicklungsmethode MechatronicUML durchgeführt werden. Zu diesem Zweck wurde eine Transformation von zeitbehafteten Controllersystemen (also dem Ergebnis der Synthese) zu MechatronicUML-Modellen umgesetzt.

Abbildung 6.1 zeigt, wie das Werkzeug zur verteilten Synthese („(Timed) Distributed Synthesis“) in diesem Prozess verwendet wird. Weiterhin wird dargestellt, welche Werkzeuge für die übrigen Prozessschritte jeweils verwendet werden. Die Abbildung zeigt außerdem die Artefakte, die Ein- oder Ausgaben der Prozessschritte sind.

Schritt 1: Im ersten Schritt modellieren die Entwickler manuell die zunächst informell vorliegenden Anforderungen an das System als MSD-Spezifikation. Dazu können sie das Werkzeug Papyrus[4] verwenden. Papyrus ist ein Eclipse-basierter Editor für UML-Modelle. ScenarioTools definiert ein UML-Profil zur Definition von MSDs, um diese als UML-Modelle spezifizieren zu können. Weiterhin erweitert ScenarioTools Papyrus durch ein Plug-in, um die Darstellung der Nachrichten durch Pfeile an die Syntax von MSDs (gestrichelt/durchgezogen für überwacht/ausgeführt, blau/rot für cold/hot) anzupassen. Durch diese Erweiterungen kann Papyrus zur Modellierung von MSD-Spezifikationen verwendet werden.

Schritt 1': Optional kann die MSD-Spezifikation nach ihrer Modellierung durch eine interaktive Schritt-für-Schritt-Ausführung („Play-out“) [BGP13] validiert werden. Diese interaktive Ausführung wird durch das Werkzeug „(Timed) MSD-Simulation“ ermöglicht. Es zeigt dem Benutzer die Nachrichten an, deren Senden den aktuellen Zustand des Systems gemäß der Spezifikation verändern würde. Aus diesen kann er dann die als

[4] https://www.eclipse.org/papyrus

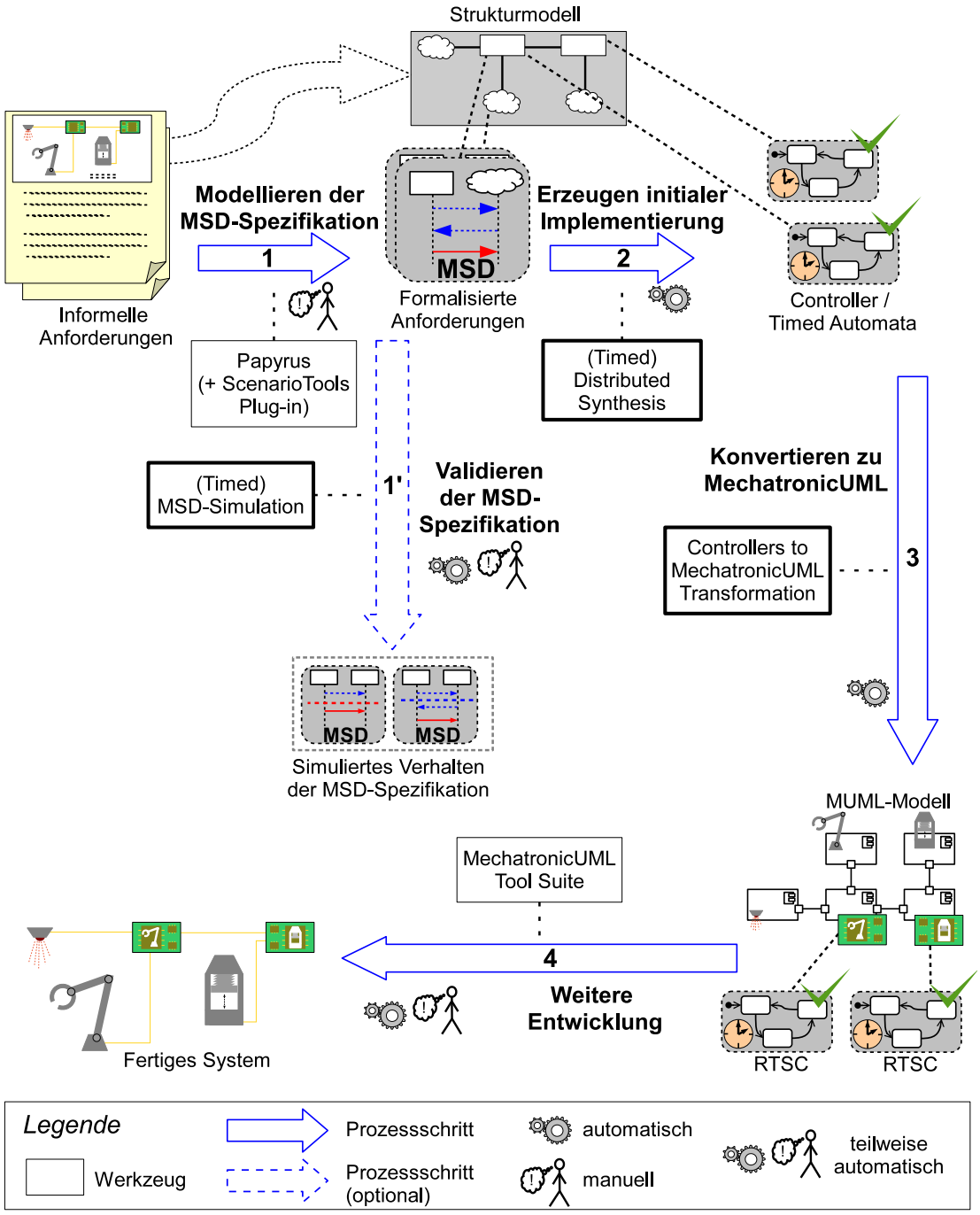


Abbildung 6.1: Prozess mit Zwischenergebnissen und verwendeten Werkzeugen

nächste zu sendende Nachricht auswählen, woraufhin die Simulation die auswählbaren Nachrichten für den resultierenden neuen Zustand anzeigt.

Während eine nicht zeitbehaftete MSD-Simulation als Teil von ScenarioTools bereits existierte, wurde sie in dieser Arbeit zu einer Echtzeitvariante (Timed MSD-Simulation, zeitbehaftetes Play-out) erweitert [BGH+14]. Diese hat den Zweck, eine interaktive Validierung auch für zeitbehaftete MSD-Spezifikationen (siehe Abschnitt 2.1.5) zu ermöglichen. Das Simulationswerkzeug wurde daher so erweitert, dass auch Zeitbedingungen und Clock Resets bei der Berechnung des Simulationszustands berücksichtigt werden (siehe Abschnitt 4.2). Auch das Benutzerinterface wurde entsprechend angepasst, beispielsweise um für den jeweils aktuellen Simulationszustand die möglichen Wertebereiche der Uhren der zeitbehafteten MSDs anzuzeigen.

Kann für die aktuell möglichen Uhrenwerte eine Zeitbedingung sowohl erfüllt, als auch nicht erfüllt sein, so kann der Benutzer auswählen, welche der beiden Möglichkeiten für den weiteren Verlauf der Simulation angenommen werden soll. Kann beispielsweise eine Uhr `c` aktuell Werte im Bereich von 5 bis 10 haben, so ist eine Zeitbedingung `c > 7` für einige mögliche Werte erfüllt ($7 < c \leq 10$), für andere jedoch nicht ($5 \leq c \leq 7$). Basierend auf der Auswahl des Benutzers wird die Simulation dann mit dem entsprechend eingeschränkten Intervall (hier also z. B. $5 \leq c \leq 7$ für die Auswahl, dass die Bedingung nicht erfüllt ist) fortgesetzt. Da heiße Zeitbedingungen mit unterer Schranke ($>$ oder $\geq$) durch Warten erfüllt werden können, wird dem Benutzer bei diesen die Auswahl ermöglicht, Zeit bis zum Erfüllen der Bedingung vergehen zu lassen. Zudem kann der Benutzer bei der Auswahl einer zu sendenden Nachricht den Sendezeitraum auf ein frei wählbares Zeitintervall einschränken.

Zusätzlich zur Echtzeitvariante der zeitbehafteten Simulation wurde ein Werkzeug zur interaktiven Zustandsraumexploration auf Basis des Editor-Frameworks Sirius[5] erstellt. Dieses bietet im Wesentlichen dieselbe Funktionalität wie die Simulation, stellt aber statt eines einzigen Zustands alle bisher erreichten Zustände des SG dar. Der Benutzer kann dadurch nicht mehr nur das im jeweils zuletzt erreichten Zustand auftretende Ereignis auswählen, sondern er kann die Auswirkungen des Auftretens beliebiger Ereignisse in *jedem* bisher erreichten Zustand untersuchen und sich den entsprechenden Folgezustand berechnen lassen. Dadurch können insbesondere alternative Ausführungspfade einfacher untersucht werden und ohne die Simulation erneut starten zu müssen.

Abbildung 6.2 zeigt einen Screenshot der Timed MSD-Simulation (oben) und der Zustandsraumexploration (unten) für das Anwendungsbeispiel der Produktionszelle. Oben links werden die im aktuellen Simulationszustand aktiven MSDs und die Objekte der Spezifikation aufgeführt. Rechts davon sind oben die Bedingungen der initialen Clock Zone zu sehen; darunter befindet sich eine Auswahlliste („Timed MessageEvent Selection View“), aus welcher der Benutzer das als nächstes auftretende Ereignis auswählen kann. Neben verschiedenen Nachrichtenereignissen kann hier auch die Möglichkeit ausgewählt werden, auf das Erfüllen einer heißen Zeitbedingung mit unterer Schranke zu warten (letzte Zeile).

Unten in der Abbildung ist die interaktive Zustandsraumexploration für dasselbe Beispiel ausschnittsweise dargestellt: Für jeden der bisher erreichten SG-Zustände werden

[5] `http://www.eclipse.org/sirius`

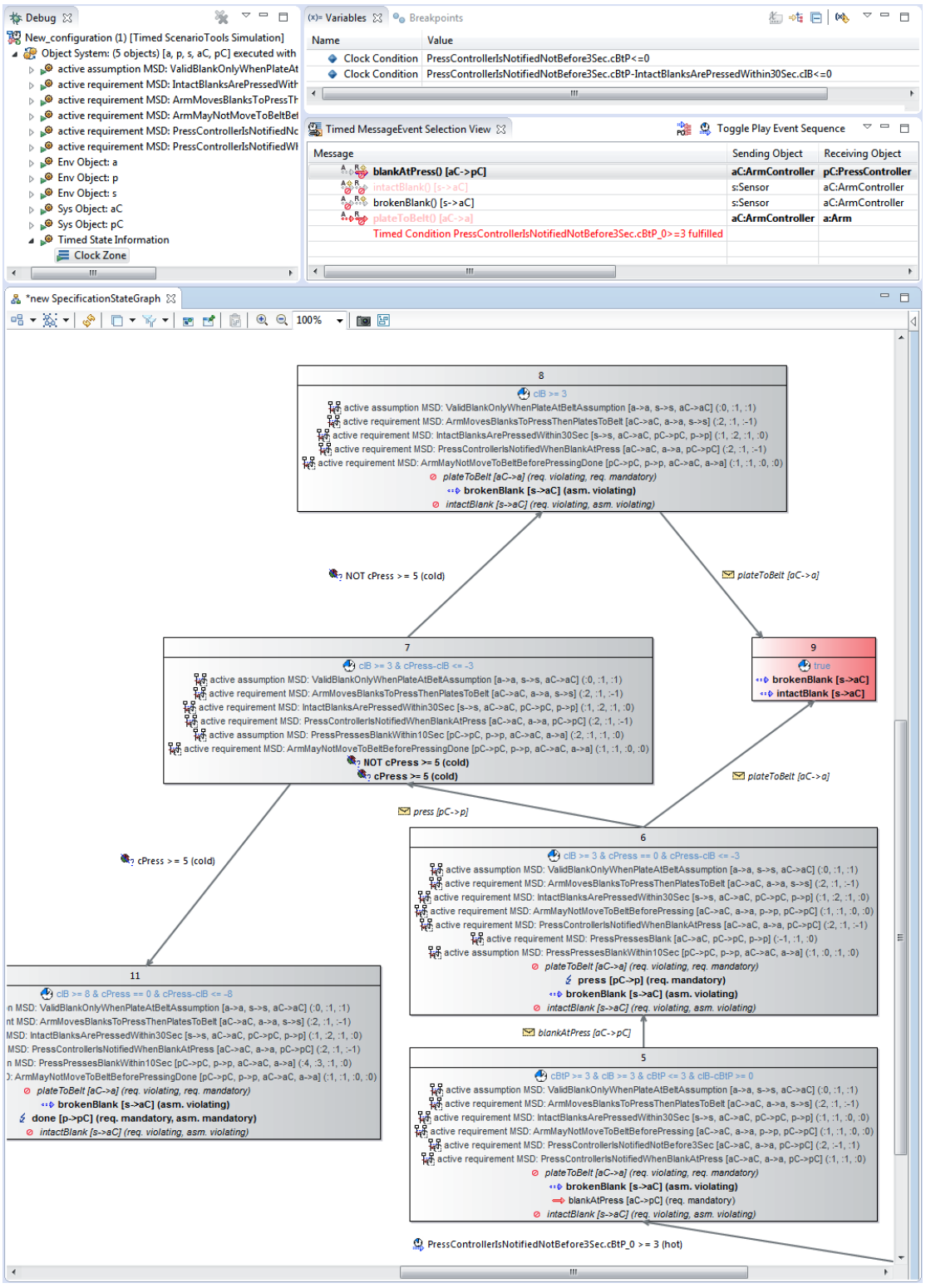


Abbildung 6.2: Screenshot der zeitbehafteten Simulationsumgebung

die initiale Clock Zone (oberste Zeile unter der Zustandsnummer), die aktiven MSDs und diejenigen Ereignisse angezeigt, die den Zustand ändern können. Hinter den Namen der aktiven MSDs stehen jeweils die Lifeline Bindings und eine Kodierung der aktuellen Position des Cuts auf den einzelnen Lifelines. Für die Ereignisse wird vermerkt, ob sie die Annahmen oder Anforderungen verletzen („violating“) bzw. in einem MSD ausgeführt („mandatory“) sind. Der Benutzer kann per Doppelklick auf ein beliebiges der Ereignisse den entsprechenden Folgezustand erzeugen lassen. An den Transitionen sind die Ereignisse eingezeichnet, die den betreffenden Übergang in den Folgezustand bewirken. Zustand 9 ist rot hervorgehoben, da bei seinem Erreichen die Anforderungen verletzt wurden.

Schritt 2: Haben sich die Entwickler davon überzeugt, dass die durch die MSD-Spezifikation formalisierten Anforderungen valide sind, so kann die eigentliche Synthese durch das Werkzeug „Distributed Synthesis“ (bzw. „Timed Distributed Synthesis“ bei zeitbehafteter MSD-Spezifikation) durchgeführt werden. Das Ergebnis dieses Schritts sind die erzeugten (ggf. zeitbehafteten) Controller (siehe Abschnitt 2.2). Dieses Werkzeug wurde in dieser Arbeit vollständig neu entwickelt. Die Einzelheiten dazu werden in Abschnitt 6.1.2 behandelt.

Schritt 3: Um die erzeugten (zeitbehafteten) Controller im Rahmen der Entwicklungsmethode MechatronicUML weiterentwickeln zu können (siehe Kapitel 5), wurde in dieser Arbeit ein Transformationswerkzeug für die Konvertierung der (zeitbehafteten) Controller in äquivalente MechatronicUML-Modelle entwickelt. Auch dies wird in Abschnitt 6.1.2 näher erläutert.

Schritt 4: Für die weiteren Schritte der Entwicklung auf Basis des automatisch erzeugten initialen MechatronicUML-Modells werden verschiedene Werkzeuge der MechatronicUML Tool Suite verwendet (siehe Abschnitt 5.2). Auf diese wird hier nicht näher eingegangen, da diese Arbeit die MechatronicUML Tool Suite selbst nicht erweitert.

6.1.2 Architektur

Abbildung 6.3 zeigt einen Überblick über die Architektur der im Rahmen dieser Arbeit entwickelten Implementierung. Neben den Bestandteilen des Syntheseverfahrens enthält die Abbildung auch die übrigen für die Verwendung der Synthese relevanten Bestandteile von ScenarioTools. Zudem ist auch weitere Software dargestellt, die nicht Teil von ScenarioTools ist, aber zur Verwendung der erstellten Implementierung benötigt wird. Die zu ScenarioTools gehörenden Bestandteile – einschließlich der in dieser Arbeit neu entwickelten Werkzeuge – sind grau hinterlegt. Die Farbe der Rechtecke ermöglicht eine Unterscheidung, ob die jeweilige Software im Rahmen der Arbeit vollständig neu entwickelt, mitentwickelt bzw. in größerem Umfang modifiziert, oder weitgehend unverändert gelassen wurde.

MSD-Runtime und MSD-Simulation Die MSD-Runtime ist eine Laufzeitumgebung für MSD-Spezifikationen. Sie verwaltet immer einen aktuellen Zustand der Spezifikation, also einen einzelnen SG-Zustand. Dieser ist definiert durch die in diesem Zustand aktiven MSDs, deren aktuelle Cuts und Variablenbelegungen, sowie die Lifeline Bindings der symbolischen Lifelines (siehe Abschnitt 2.1.2). Basierend auf dem aktuellen Zustand ermittelt die MSD-Runtime, welche Ereignisse in diesem Zustand zu einer Zustandsänderung führen [BGP13]. Für ein gegebenes Ereignis berechnet sie den Folgezustand der Spezifikation, der dann der neue aktuelle Zustand ist. Weiterhin kann die MSD-Runtime zum schrittweisen Aufbauen eines SG verwendet werden (siehe Abschnitt 3.2.1), der alle bis zum jeweiligen Zeitpunkt erreichten Zustände enthält. Dabei werden Duplikate erkannt und zu einem einzigen Zustand zusammengefasst. Insbesondere diese Erzeugung des SG wurde in dieser Arbeit mitentwickelt. Die in Abschnitt 6.1.1 aus Benutzersicht genauer beschriebene MSD-Simulation ist ein Frontend zur MSD-Runtime.

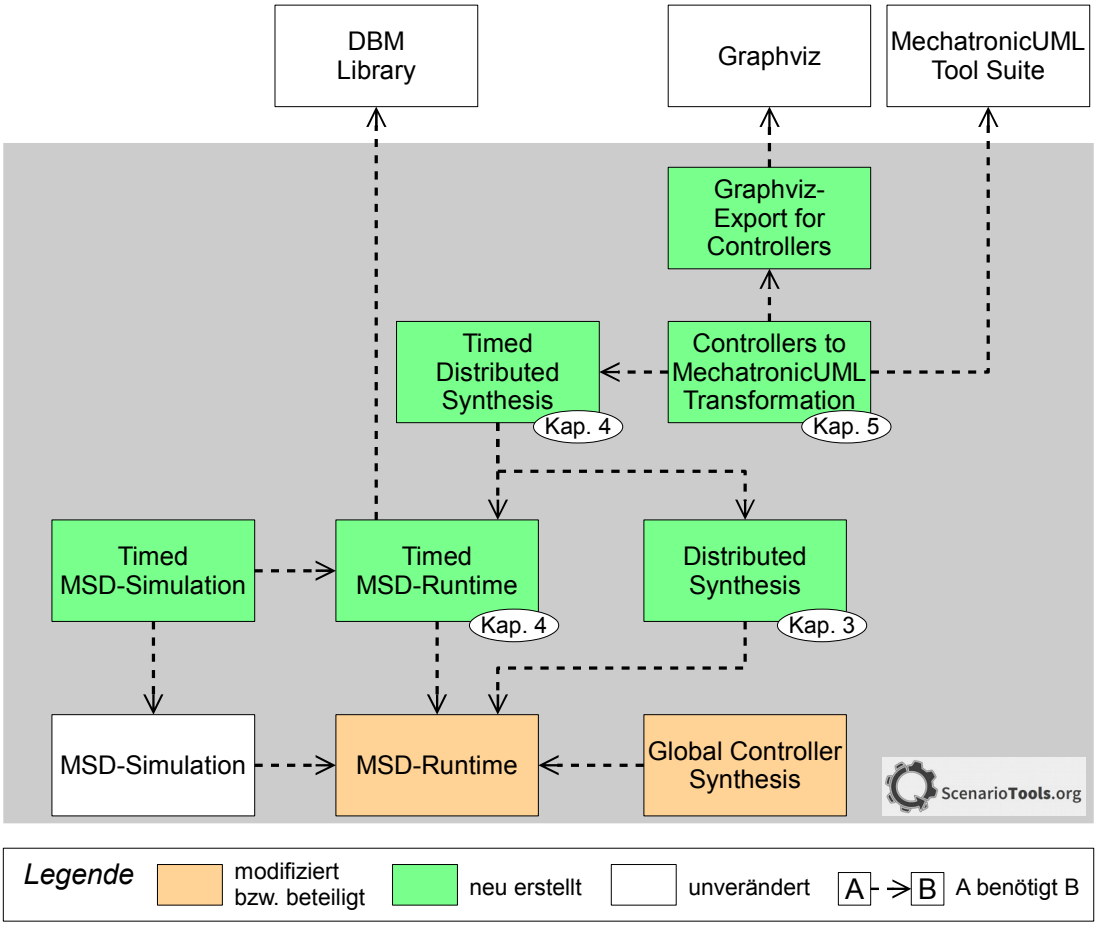


Abbildung 6.3: Architektur der Implementierung

Timed MSD-Runtime und Timed MSD-Simulation Um auch die Simulation und Synthese von zeitbehafteten MSD-Spezifikationen, die aus Timed MSDs (siehe Abschnitt 2.1.5) bestehen, zu ermöglichen, wurde in dieser Arbeit die „Timed MSD-Runtime“ als Echtzeitvariante der MSD-Runtime entwickelt [BGH+14]. Diese setzt die in Abschnitt 4.2 beschriebene Erzeugung von Folgezuständen im zeitbehafteten SG und die Ermittlung der in einem Zustand möglichen Ereignisse um.

Wie in Abschnitt 4.2 genauer beschrieben ist, wird Zeit in der zeitbehafteten MSD-Runtime durch die Modellierung der bei Betreten jedes Spezifikationszustands möglichen Uhrenwerte mittels Clock Zones [BY04] berücksichtigt. Dabei wird für die Ausführung der grundlegenden Berechnungsoperationen auf Clock Zones eine bereits bestehende Java-Implementierung der von Bengtsson et al. beschriebenen Difference Bound Matrices (DBMs) [BY04] verwendet. Diese Implementierung basiert auf dem von Eckardt und Heinzemann [EH11] beschriebenen Framework und wurde vom zweiten Autor dieser Veröffentlichung in Form der „DBM Library“ entwickelt.

Für die nicht zeitbehafteten Anteile des MSD-Verhaltens wird weiterhin auf die ursprüngliche MSD-Runtime zurückgegriffen. Auf Basis der Timed MSD-Runtime wurde die bereits im vorhergehenden Abschnitt diskutierte Timed MSD-Simulation als Erweiterung der bereits existierenden MSD-Simulation entwickelt.

Die Timed-MSD-Runtime erweitert die MSD-Runtime nicht nur um Echtzeit, sondern implementiert auch die in Abschnitt 4.4 beschriebene asynchrone Kommunikation mit Zeitbedarf. Dabei werden in jedem Zustand die aktuell in Übertragung befindlichen Nachrichten in einer Queue verwaltet.

Distributed Synthesis Die in Kapitel 3 beschriebene verteilte Synthese wurde in Form des Werkzeugs „Distributed Synthesis“ realisiert. Dabei wird für die Erzeugung des SG auf die MSD-Runtime zurückgegriffen. Die Synthese kann für ein spezielles EMF-Modell aufgerufen werden, das die MSD-Spezifikation definiert. Bei Erfolg der Synthese werden die erzeugten Controller zur späteren Verwendung ebenfalls in einem EMF-Modell gespeichert. In jedem Fall wird nach der Synthese ein Dialogfenster mit dem Syntheseergebnis (erfolgreich oder nicht) angezeigt. Weiterhin werden Informationen wie die Größe des explorierten SG, die benötigte Zeit, etc. angezeigt, die für die Evaluierung des Verfahrens relevant sind (siehe Abschnitt 6.2.2).

Für die Implementierung der Synthese wurde XCore[6] verwendet, eine textuelle Syntax für EMF. Gegenüber dem bekannteren ECore hat XCore unter anderem den Vorteil, dass die auf dem Modell definierten Operationen mittels XBase[7] direkt in XCore implementiert werden können. XBase ist eine Java-ähnliche Sprache, die jedoch typischerweise kompakteren Code ermöglicht und die zu Java-Code compiliert wird.

Timed Distributed Synthesis Die in Kapitel 4 beschriebene zeitbehaftete Erweiterung der Synthese wurde in Form des Werkzeugs „Timed Distributed Synthesis“ realisiert. Dieses verwendet statt der ursprünglichen MSD-Runtime die neu entwickelte zeitbehaftete Variante zur Erzeugung des SG. Da die Synthese aber auch direkt Clock Zones

[6] `https://wiki.eclipse.org/Xcore`
[7] `https://wiki.eclipse.org/Xbase`

manipulieren muss (siehe Abschnitt 4.3.3), verwendet sie intern eine Variante der von Bengtsson et al. beschriebenen DBMs [BY04]. Diese wurde auf Basis von EMF in dieser Arbeit entwickelt, um Kompatibilität zu den übrigen EMF-Modellen zu erreichen.

Controllers to MechatronicUML Transformation Das im vorhergehenden Abschnitt erwähnte Werkzeug zur Konvertierung von Controllern zu MechatronicUML-Modellen verwendet als Eingabe das durch die Synthese erzeugte Modell eines zeitbehafteten Controllersystems. Als zusätzliche Eingabe wird das Strukturmodell der MSD-Spezifikation verwendet, für die die Synthese ausgeführt wurde. Die Ausgabe der Transformation ist ein MechatronicUML-Modell, das insbesondere Real-Time Statecharts (RTSCs) enthält, die dasselbe Verhalten wie die gegebenen Controller definieren (siehe Abschnitt 5.1.1). Wie in Abschnitt 5.1.2 beschrieben wurde, werden neben der Ableitung von RTSCs aus Controllern auch Komponenten und Komponenteninstanzen für das gegebene Strukturmodell erzeugt. Die in Abschnitt 5.1.3 beschriebene Anpassung wurde jedoch nicht implementiert, sondern muss bisher manuell ausgeführt werden.

Zusätzlich zu den erzeugten Modellen werden entsprechende Diagramme generiert, welche die erzeugten Modellelemente beinhalten. Um die Zustände der als Teil des MechatronicUML-Modells erzeugten RTSCs im betreffenden Diagramm automatisch anzuordnen, wird das Werkzeug Graphviz[8] verwendet. Die Controller werden dabei zunächst durch das neu entwickelte Werkzeug „Graphviz Export for Controllers“ in das Graphviz-Eingabeformat überführt. Graphviz wird dann zur automatischen Erzeugung von Koordinaten für die Zustände des Controllers verwendet, die für die entsprechenden Zustände des erzeugten RTSCs im Diagramm übernommen werden. Ohne dieses automatische Layout würde eine manuelle Weiterentwicklung der erzeugten RTSCs eine manuelle Anordnung aller Zustände, Transitionen und Beschriftungen erfordern. Dies kann abhängig von der Größe des Controllers sehr zeitaufwendig sein.

Weitere Werkzeuge Das bereits erwähnte Werkzeug „Graphviz Export for Controllers“ kann auch zur Generierung von PDF-Dateien aus (zeitbehafteten) Controllern und SG verwendet werden. Der betreffende Graph wird dabei durch das Werkzeug Graphviz automatisch angeordnet und in eine PDF-Datei geschrieben. Das erzeugte Layout vermeidet Überlappungen von Knoten, Kanten und Beschriftungen und ist meist ausreichend übersichtlich, um die erzeugten Controller verstehen zu können.

Wie in Abschnitt 3.2.4 erwähnt wurde, wurde vom Autor dieser Arbeit im Rahmen einer Kooperation auch ein neuer Synthesealgorithmus für globale Controller mitentwickelt [GBC+13] („Global Controller Synthesis“). Dieser Ansatz basiert ebenfalls auf der (nicht zeitbehafteten) MSD-Runtime. Dieser Syntheseansatz kann zur Erzeugung einer zentralen Implementierung einer MSD-Spezifikation verwendet werden und damit zur Prüfung, ob die MSD-Spezifikation frei von widersprüchlichen Anforderungen ist.

[8] `http://www.graphviz.org`

6.2 Evaluierung

Unter Verwendung der im vorherigen Abschnitt beschriebenen Implementierungen wurde das entwickelte Syntheseverfahren durch Anwendung auf verschiedene Beispiele evaluiert. Diese umfassen verschiedene Versionen des in Abschnitt 1.1 eingeführten Anwendungsbeispiels der Produktionszelle und weitere Beispielspezifikationen ähnlicher Größe. Diese Beispiele dienen zur Untersuchung der grundsätzlichen Anwendbarkeit des Verfahrens und seiner Erweiterungen. Zur Evaluierung der Skalierbarkeit des Ansatzes wurden erweiterbare Beispielspezifikationen modelliert, deren Größe sich mit wenig Aufwand variieren lässt.

Die grundsätzliche Eignung von MSDs zur Spezifikation des Systemverhaltens mechatronischer Systeme wurde bereits mit positivem Ergebnis evaluiert [Gre11]. Auch Details, wie beispielsweise der Nutzen von Assumption MSDs und bestimmten Modellelementen, wurden dort diskutiert. Daher werden diese Aspekte hier ausgeklammert und es wird ausschließlich die Verwendbarkeit der Synthesetechnik untersucht.

In Abschnitt 6.2.1 werden zunächst die verschiedenen Anwendungsbeispiele diskutiert. Die Evaluierung der Skalierbarkeit wird in Abschnitt 6.2.2 behandelt.

6.2.1 Evaluierung anhand des Anwendungsbeispiels der Produktionszelle und weiterer Beispiele

Das implementierte Syntheseverfahren wurde auf das in dieser Arbeit verwendete durchgehende Anwendungsbeispiel (siehe Abschnitt 1.1) der Produktionszelle angewendet, um die grundsätzliche Funktionalität des Syntheseansatzes zu untersuchen. In diesem Beispiel wurden die Anforderungen an das Systemverhalten zunächst durch die in Abschnitt 3.1 vorgestellte MSD-Spezifikation modelliert.

Um die Anwendbarkeit von Erweiterungen des Verfahrens untersuchen zu können wurde das Anwendungsbeispiel mehrfach erweitert: In Abschnitt 3.3.7 wurde eine Variante des Beispiels beschrieben, bei der zwei Synchronisationsnachrichten statt nur einer erforderlich sind. In Abschnitt 3.4.1 wurde das Beispiel um strukturelle Einschränkungen der möglichen Kommunikation erweitert. In Abschnitt 4.1 wurde eine Variante mit Echtzeitanforderungen diskutiert, in Abschnitt 4.4.2 wurde schließlich asynchrone Kommunikation mit Zeitbedarf in diese Echtzeitvariante des Beispiels eingeführt.

Die zur Evaluierung verwendeten MSD-Spezifikationen der Produktionszelle stellen keine Spezifikation realer Systemanforderungen dar. Allerdings orientieren sie sich an einem praxisnahen Beispiel, das in der Literatur aufgrund seiner Realitätsnähe vorgeschlagen wurde [LL95]. Die Evaluierung bestand in der Ausführung des Verfahrens (bzw. der Echtzeitvariante davon im Fall der zeitbehafteten Spezifikationen) auf jede Variante des Beispiels und in einer manuellen Überprüfung des Ergebnisses der Synthese auf Einhaltung der MSD-Spezifikation. Die Synthese konnte jeweils in deutlich unter einer Sekunde eine korrekte Implementierung erzeugen.

Zusätzlich wurde das Verfahren auf einige bestehende Beispiel-Spezifikationen angewendet, die im ScenarioTools-Projekt entstanden sind. Dazu gehören einerseits einige abstrakte Beispiele, die keinem realen System entsprechen. Andererseits gehören dazu verschiedene Szenarien für das RailCab-System [HSD+15]. Das RailCab-System ist ein

Konzept für ein verteiltes mechatronisches System aus autonom fahrenden Schienenfahrzeugen zum Transport von Personen oder Gütern.

Die abstrakten Beispiele sind zwar nicht geeignet, eine praktische Anwendbarkeit des Verfahrens zu belegen, sie enthalten jedoch oft bestimmte Spezial- oder Problemfälle, die in kleineren Beispielspezifikationen für reale Systeme schwer zu konstruieren sind. Da diese Problemfälle dennoch in der Realität auftreten können, sind diese Beispiele also durchaus hilfreich: Durch die erfolgreiche Anwendung der verteilten Synthese auf die abstrakten Beispiele konnte gezeigt werden, dass diese auch für die betreffenden Spezialfälle funktioniert, nicht nur für die „typische“ MSD-Spezifikation.

Die Anwendung des Verfahrens auf die Beispiele im Kontext des RailCab-Systems hatte ein anderes Ziel: Wie beim durchgehenden Beispiel der Produktionszelle sollte hierdurch die Anwendbarkeit des Verfahrens auf ein reales mechatronisches System belegt werden. Auch hier wurden für die Beispiele korrekte verteilte Controller erzeugt.

6.2.2 Evaluierung der Skalierbarkeit

Zur Untersuchung der Skalierbarkeit des Syntheseverfahrens wurde ein zusätzliches Anwendungsbeispiel entwickelt, dessen Größe sich in gleichmäßigen Schritten steigern lässt. Durch Anwendung der Synthese auf Varianten des Beispiels von unterschiedlicher Größe kann so der Zusammenhang der Laufzeit des Verfahrens mit der Größe der gegebenen MSD-Spezifikation untersucht werden.

Als Grundlage für dieses Evaluierungsbeispiel wurde erneut das durchgängige Beispiel der Produktionszelle (siehe Abschnitt 1.1) verwendet. Nachfolgend wird zunächst erläutert, wie dieses Beispiel angepasst wurde, um es systematisch erweiterbar zu machen. Anschließend werden die Ergebnisse von Laufzeitmessungen diskutiert, die bei der Ausführung der Synthese für das Beispiel durchgeführt wurden.

Evaluierungsbeispiel zur Untersuchung der Skalierbarkeit

Als Grundlage für das Evaluierungsbeispiel wurde die Beispiel-Spezifikation aus Abschnitt 3.1 verwendet. Für die Evaluierung wurde diese Spezifikation erweitert, um die Komplexität des spezifizierten Systems und damit den zur Synthese erforderlichen Aufwand auf einfache Weise anpassen zu können. Als Maß für die Systemkomplexität wird hier die Größe des Spezifikationsgraphen (SG) betrachtet, der durch die MSD-Spezifikation definiert wird. Eine Vergrößerung des SG wird durch eine schrittweise Erhöhung der Anzahl der Objekte im Beispiel erreicht.

Wie die Basisvariante des Beispiels modelliert auch das Evaluierungsbeispiel eine Produktionsanlage, die über ein Förderband ankommende Rohlinge zu Metallplatten presst. Auch hier bewegt ein Roboterarm zunächst die Rohlinge zu einer Presse. Der Roboterarm legt die fertig gepressten Metallplatten jedoch nicht auf ein Ablageförderband, sondern auf eine Ablagefläche, von der sie ein weiterer Roboterarm aufnehmen und zu einer weiteren Presse bewegen kann (und von dort zu einem weiteren Arm, etc.). Auf diese Weise können beliebig viele Roboterarme verkettet werden, denen jeweils eine eigene Presse zugeordnet ist. Jede Presse und jeder Roboterarm werden von einem eigenen Controller gesteuert, der durch die Synthese erzeugt werden muss.

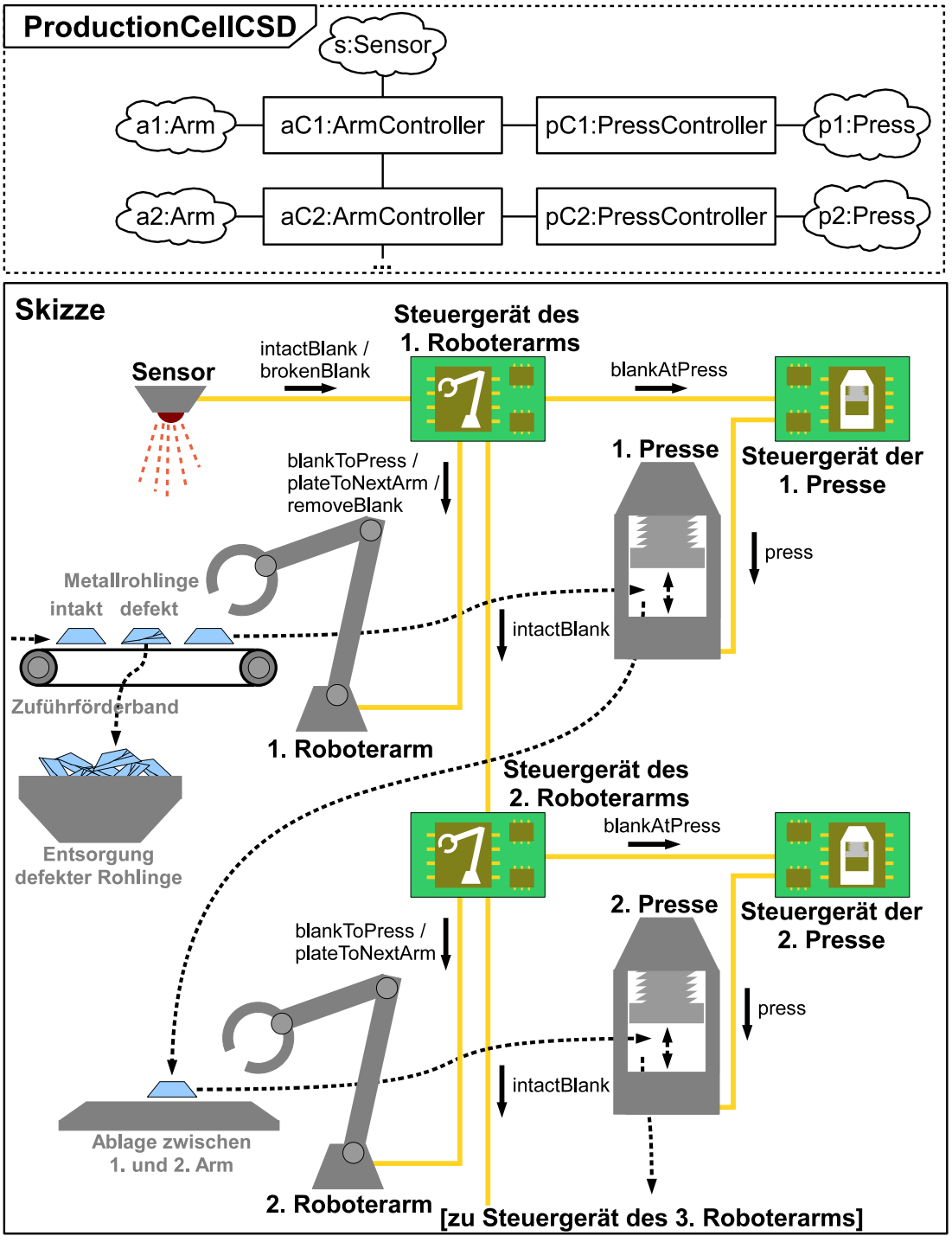


Abbildung 6.4: Struktur des Systems auf Instanzebene als Kompositionsstrukturdiagramm (oben) und als Skizze (unten)

Abbildung 6.4 zeigt die Struktur des Anwendungsbeispiels exemplarisch für zwei Produktionszellen (das Beispiel enthält mindestens eine weitere, die nicht dargestellt ist). Oben sind die Instanzen der involvierten Subsysteme in einem Kompositionsstrukturdiagramm dargestellt, unten zeigt eine Skizze diese Subsysteme und zusätzliche Bestandteile der Umgebung. Weiterhin zeigt die Skizze auch die über die Kommunikationsverbindungen ausgetauschten Nachrichten. Die Struktur auf Typebene ist dieselbe wie in Abschnitt 3.1 und wird hier deshalb nicht wiederholt. Auch die möglichen Nachrichten sind dieselben, mit Ausnahme der Ersetzung von `plateToBelt` durch `plateToNextArm`.

Nur das Steuergerät des ersten Arms empfängt direkt Nachrichten von der Umgebung – in diesem Fall vom Sensor am Zuführförderband. Die Steuergeräte der übrigen Arme werden jeweils vom Steuergerät des vorhergehenden Arms durch die Nachricht `intactBlank` darüber unterrichtet, dass eine neue Metallplatte (bzw. aus Sicht des nächsten Arms ein Rohling) abgelegt wurde, die weiterverarbeitet werden soll.

In diesem Beispiel wird angenommen, dass nur die über das Zuführförderband ankommenden Rohlinge defekt sein können, während die Metallplatten zwischen den Pressvorgängen und während dieser nicht beschädigt werden können. Daher muss nur der erste Roboterarm defekte Rohlinge aussortieren, womit dieser Fall in den Controllern der übrigen Roboterarme nicht behandelt werden muss.

Die MSDs der Spezifikation wurden größtenteils aus Abschnitt 3.1 übernommen, jedoch mit geringen Änderungen: Damit nicht für jede Größenvariante des Beispiels neue MSDs erstellt werden müssen, wurden die meisten konkreten Lifelines der ursprünglichen Spezifikation in symbolische Lifelines (siehe Abschnitt 2.1.4) umgewandelt. Sie beziehen sich also nicht mehr auf statisch vorgegebene Instanzen, sondern schränken lediglich den Typ des jeweils repräsentierten Objekts ein. Weitere Einschränkungen der repräsentierten Instanzen ergeben sich ggf. durch Binding Expressions (ebenda), die in den betreffenden MSDs ergänzt wurden, wo dies erforderlich war.

Da das Steuergerät des ersten Arms ein Spezialfall ist (es empfängt als einziges Nachrichten vom Sensor), beziehen sich einige Anforderungen nur auf dieses. In den betreffenden MSDs wurden für den ersten Arm und sein Steuergerät konkrete Lifelines beibehalten. Dies gilt auch für den Sensor, den es nur einmal gibt. Das MSD ArmRemovesBrokenBlank aus Abschnitt 3.1.2 bleibt daher unverändert. Aus dem MSD ArmMovesBlanksToPressThenPlatesToBelt der ursprünglichen Spezifikation wurden jedoch zwei MSDs in der Spezifikation des Evaluierungsbeispiels. Diese sind in Abbildung 6.5 dargestellt: FirstArmMovesBlanksToPressThenPlatesToNextArm definiert die Reaktion des ersten Arms auf die `intactBlank`-Nachricht des Sensors, während ArmMovesBlanksToPressThenPlatesToNextArm die Übergabe einer Metallplatte von einem beliebigen Roboterarm zum nachfolgenden Roboterarm und dessen Reaktion darauf modelliert.

Das MSD ArmMovesBlanksToPressThenPlatesToNextArm wird aufgerufen, wenn ein ArmController aC seinen Arm per `plateToNextArm` anweist, die Metallplatte (nach dem Pressen) weiterzugeben. Eine kalte Bedingung stellt sicher, dass das MSD sofort wieder beendet wird, wenn es sich bei aC um den letzten ArmController handelt. Andernfalls werden durch die Binding Expressions an den übrigen Lifelines diese an den nächsten ArmController und dessen Arm gebunden. Dann muss aC an seinen Nachfolger nextAC `intactBlank` senden, worauf dieser wie im MSD ArmMovesBlanksToPressThenPlatesToBelt durch Senden entsprechender Nachrichten an seinen Arm nextA reagieren muss.

Auf die Darstellung der übrigen MSDs der Spezifikation wird hier verzichtet, da sie im Wesentlichen denen aus Abschnitt 3.1 mit den diskutierten Änderungen entsprechen.

Um den Berechnungsaufwand für die Clock Zones bei der zeitbehafteten Synthese zu untersuchen, wurde das Evaluierungsbeispiel in einer geringfügig erweiterten Variante um die in Abbildung 6.5 bereits eingezeichneten Zeitbedingungen erweitert. Die erste Zeitbedingung fordert dabei eine minimale Wartezeit zwischen zwei Nachrichten, während die zweite Zeitbedingung eine maximale Zeitspanne für das Senden mehrerer Nachrichten definiert.

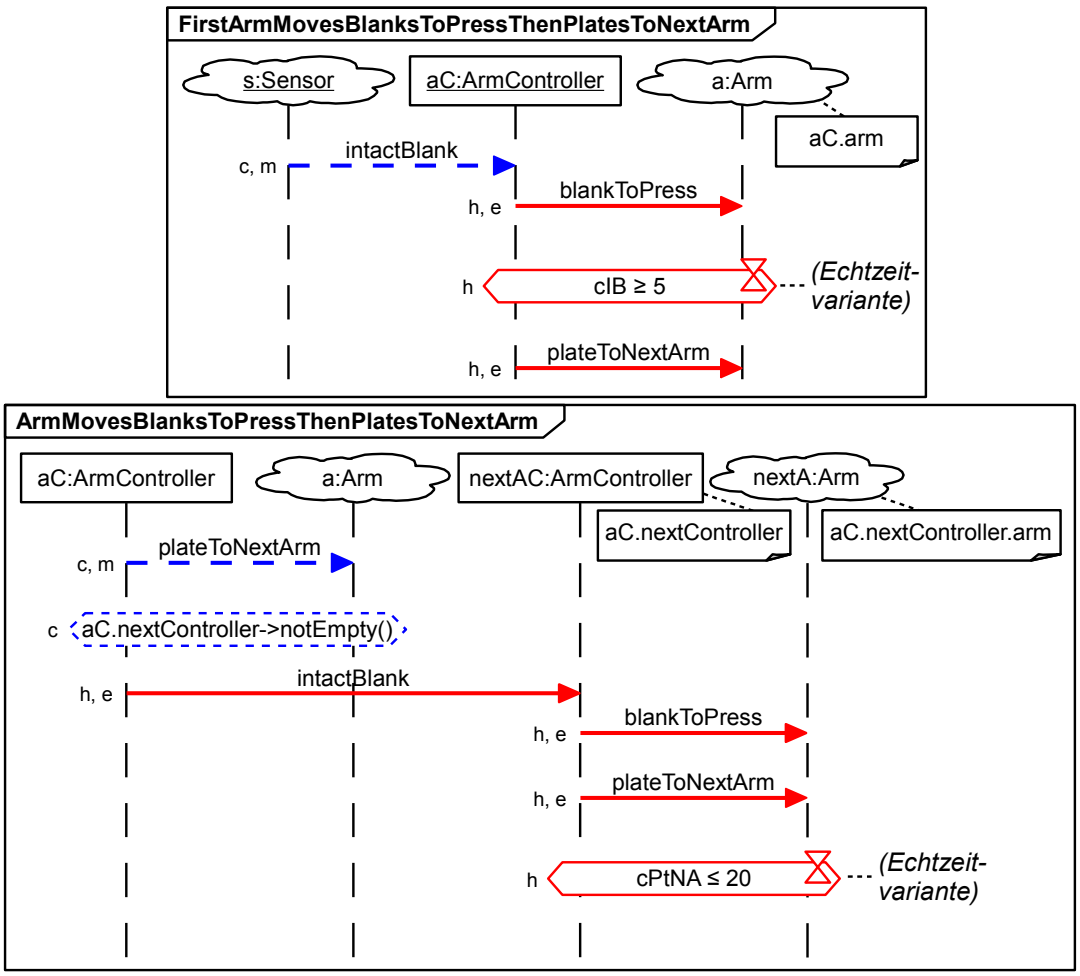


Abbildung 6.5: Zwei der für das Evaluierungsbeispiel geänderten MSDs

Auswertung und Diskussion

Die in Abschnitt 6.1 beschriebene Implementierung des Synthesealgorithmus wurde auf mehrere Größenstufen des beschriebenen Evaluierungsbeispiels angewendet – jeweils mit und ohne Echtzeitanforderungen. Tabelle 6.1 zeigt die Ergebnisse.

Dabei wurde die Anzahl der Arme und Pressen variiert, wobei diese im Beispiel jeweils paarweise auftreten und es von beiden daher immer eine identische Anzahl gibt. Die Anzahl der zu erzeugenden Controller ist daher immer gerade. Es gibt etwa doppelt so viele Objekte wie Controller, da für jede Instanz von ArmController ein Arm und für jeden PressController eine Press existiert, sowie immer insgesamt ein Sensor.

Für jede Ausbaustufe der Beispielspezifikation bis 20 Controllern wurde die Synthese mehrfach angewendet, um genauere Werte ermitteln zu können. Bei jedem der 10 Durchläufe wurde die Laufzeit gemessen. Aus diesen Ergebnissen wurde dann jeweils das arithmetische Mittel aller Durchläufe gebildet. Bei den Beispielen mit 30 und 40 Controllern wurde jeweils nur ein Durchlauf durchgeführt. Die Messungen wurden unter Verwendung einer Intel Core i5-2500K CPU mit 3.3 GHz durchgeführt. Zusätzlich wurde die Anzahl berechneter Controllersystem-Kandidaten und die Größe des SG in Zuständen und Transitionen ermittelt. Für die Echtzeitvariante des Evaluierungsbeispiels wurde dieselbe Evaluierung erneut durchgeführt.

Tabelle 6.1: Ermittelte Werte für das Evaluierungsbeispiel

		ohne Zeitbedingungen				*mit Zeitbedingungen*			
Controller	**Objekte**	**SG-Zustände**	**Transitionen**	**Kandidaten**	**Laufzeit**	**SG-Zustände**	**Transitionen**	**Kandidaten**	**Laufzeit**
2	5	7	8	9	41 ms	9	11	11	69 ms
4	9	12	14	17	67 ms	15	18	20	139 ms
6	13	17	20	25	209 ms	21	25	30	428 ms
8	17	22	26	33	493 ms	27	32	40	1,2 s
10	21	27	32	41	1,3 s	33	39	50	2,9 s
12	25	32	38	49	2,7 s	39	46	60	5,0 s
14	29	37	44	57	4,5 s	45	53	70	10,8 s
16	33	42	50	65	8,0 s	51	60	80	22,4 s
18	37	47	56	73	15,9 s	57	67	90	33,2 s
20	41	52	62	81	27,4 s	63	74	100	56,0 s
30	61	77	92	121	2 min 53 s	93	109	150	8 min 57 s
40	81	102	122	161	16 min 13 s	123	144	200	41 min 14 s

Die Ergebnisse zeigen, dass die Laufzeit des Verfahrens für das gewählte Beispiel exponentiell von der Anzahl der Controller abhängt. Sie zeigen jedoch auch, dass für kleine Beispiele eine Synthese innerhalb weniger Sekunden oder Minuten möglich ist. Sie ist damit für Systeme dieser Größe deutlich schneller als die manuelle Implementierung und zusätzliche Verifikation, die sie ersetzt. Die Synthese der zeitbehafteten Variante des Beispiels benötigte meist etwas mehr als doppelt so lange wie die nicht zeitbehaftete (und für 30 und 40 Controller deutlich länger), war aber für das Beispiel in den betrachteten Größen ebenfalls in vertretbarer Zeit möglich.

6.3 Zusammenfassung

Dieses Kapitel behandelte zunächst die Implementierung der Konzepte aus den Kapiteln 3, 4 und 5. Diese Implementierung besteht aus mehreren Eclipse-Plug-ins. Insbesondere wurden das Syntheseverfahren, dessen Echtzeitvariante und ein Werkzeug zur Konvertierung von Controllern zu MechatronicUML-Modellen implementiert. Weiterhin wurde auch das Simulationswerkzeug der Eclipse-basierten Werkzeugumgebung ScenarioTools in dieser Arbeit um Echtzeitverhalten erweitert.

Die Evaluierung des Syntheseansatzes wurde zunächst anhand einiger Beispielspezifikationen durchgeführt. Zur Evaluierung der Laufzeit des Verfahrens und von dessen Skalierbarkeit wurde ein spezielles Evaluierungsbeispiel verwendet. Dieses Beispiel und die für dieses erzielten Ergebnisse wurden hier ebenfalls vorgestellt. Das Verfahren skaliert (in der implementierten Form) nicht gut für eine steigende Anzahl zu erzeugender Controller. Für die relativ kleinen untersuchten Beispiele war der Zeitaufwand aber akzeptabel für eine praktische Anwendung, da der manuelle Aufwand für dieselbe Aufgabe deutlich höher ist – und das Resultat nicht notwendigerweise die Spezifikation einhält.

Kapitel 7

Verwandte Arbeiten

Dieses Kapitel behandelt andere Arbeiten, die zu dem im Rahmen dieser Dissertation erarbeiteten Syntheseansatz verwandt sind. In Abschnitt 7.1 werden zunächst Arbeiten behandelt, die ebenfalls die Synthese verteilter Systeme betrachten und damit ein ähnliches Problem lösen wie das in Kapitel 3 beschriebene Verfahren. Abschnitt 7.2 behandelt Ansätze für zeitbehaftete Synthese, die zur zeitbehafteten Erweiterung der verteilten Synthese in Kapitel 4 verwandt sind. Schließlich fasst Abschnitt 7.3 die wesentlichen Unterschiede dieser Arbeit zu den diskutierten verwandten Arbeiten zusammen.

7.1 Verteilte Synthese

Grundsätzlich kann bei den verteilten Syntheseansätzen zwischen Verfahren unterschieden werden, die zusätzliche Synchronisationsnachrichten einfügen, und solchen, die dies nicht tun. Da der in dieser Arbeit entwickelte Syntheseansatz zu ersteren gehört, kann diese Gruppe von Ansätzen als näher verwandt betrachtet werden. Eine dritte Alternative ist, durch Einschränkungen der Spezifikationssprache zu garantieren, dass keine zusätzlichen Nachrichten erforderlich sind. Bontemps und Heymans [BH05] bieten einen systematischen Überblick über Syntheseansätze auf Basis von Live Sequence Charts (LSCs), der auch verteilte Ansätze behandelt. Die in dieser Arbeit betrachteten Modal Sequence Diagrams (MSDs) sind den LSCs sehr ähnlich. Generell gilt für alle verwandten Arbeiten zur verteilten Synthese, dass diese keine Echtzeitsysteme betrachten.

Abschnitt 7.1.1 behandelt Ansätze mit zusätzlicher Synchronisation, während Abschnitt 7.1.2 solche ohne behandelt. Abschnitt 7.1.3 diskutiert einen Ansatz, bei dem bereits die Eingabesprache den Fall ausschließt, dass Synchronisationsnachrichten erforderlich sind.

7.1.1 Ansätze mit zusätzlicher Synchronisation

Die folgenden Ansätze fügen dem System zusätzliches Kommunikationsverhalten hinzu, das nicht bereits in der Spezifikation definiert ist. Das Ziel der zusätzlichen Kommunikation ist dasselbe wie in dieser Arbeit: Sie ermöglicht Subsystemen, zwischen erforderlichem und verbotenem Systemverhalten zu unterscheiden, wenn diese dazu selbst unzureichende Informationen über den globalen Zustand des Systems haben. Die zusätzlich erforderlichen Informationen werden dann von anderen Subsystemen durch Kommunikation erlangt.

Erzeugung vollständig synchronisierter Kopien eines globalen Controllers

Harel und Kugler [HK02] beschreiben einen Ansatz für die Synthese verteilter Controller auf Grundlage von LSC-Spezifikationen. Der Ansatz erzeugt zunächst einen globalen Controller, also einen Automaten, der ein gemäß der Spezifikation korrektes Verhalten aller Subsysteme beschreibt. Eine verteilte Implementierung wird konstruiert, indem für jedes Subsystem eine Kopie dieses globalen Controllers erstellt wird und indem zusätzliche Nachrichten eingeführt werden, die diese Kopien synchron halten.

Im Gegensatz zu dem in dieser Arbeit vorgestellten Syntheseansatz erzeugt dieser Ansatz jedoch auch dann zusätzliche Synchronisationsnachrichten, wenn das lokale Wissen der Subsysteme für ein korrektes Verhalten gemäß der Spezifikation bereits ausreicht. Die unnötigerweise zusätzlich eingeführten Nachrichten vermindern generell die Performanz des verteilten Systems und können – insbesondere bei einer hohen Anzahl von Subsystemen – dazu führen, dass das verwendete Netzwerk überlastet wird. Im Fall von Echtzeitsystemen – die durch den Ansatz nicht betrachtet werden – kann das auch zur Verletzung von Echtzeitanforderungen führen. Zudem setzt der Ansatz voraus, dass alle Subsysteme miteinander kommunizieren können.

In Abschnitt 3.2.4 wurden bereits einige grundsätzliche Nachteile einer nachträglichen Verteilung eines globalen Controllers erwähnt. Da auch bei diesem Ansatz die Verteilung erst in einem separaten Schritt unabhängig von der Erzeugung des globalen Controllers erfolgt, gelten diese Nachteile auch hier.

In derselben Veröffentlichung wird weiterhin eine Variante mit nur partieller Duplikation des globalen Controllers diskutiert, bei welcher auch der Synchronisationsaufwand geringer ist. Dabei findet jedoch keine Modellierung der Informationen der Subsysteme über die möglichen globalen Zustände statt. Daher verbleiben viele Fälle, in denen unnötige Synchronisationsnachrichten dem empfangenden Subsystem Informationen übermitteln, über die es bereits verfügt.

Einschränken bestehender Implementierungen zur Einhaltung neuer Anforderungen

Katz et al. [KPS11] und Peled und Schewe [PS11] modellieren das lokale Wissen der Subsysteme über den globalen Gesamtzustand auf ähnliche Weise wie der in dieser Arbeit vorgestellte Syntheseansatz. Weiterhin verwenden sie ebenfalls Synchronisationen zum Austausch von Informationen zwischen Subsystemen, wenn das lokale Wissen nicht ausreicht. In einer neueren Arbeit von Peled und Schewe [PS14] werden neben Safety-Anforderungen auch andere Typen von Anforderungen betrachtet, darunter auch „Repeated Reachability“, also ein unendlich häufiges Erreichen spezieller Zustände. Mit dieser Erweiterung könnte das Verfahren grundsätzlich auch für die Anwendung auf MSD-Spezifikationen angepasst werden, da auch hier die wiederholte Erreichbarkeit von Zuständen mit bestimmten Eigenschaften – den Zielzuständen – sichergestellt werden muss (siehe Abschnitt 2.1.6).

Keiner dieser Ansätze führt jedoch eine Synthese von komplett neuem Verhalten auf Grundlage einer Spezifikation durch. Stattdessen beginnen sie mit einem bereits *existierenden* Implementierungsmodell eines verteilten Systems, das sie als Eingabe in Form eines Petrinetzes erhalten. Eine weitere Eingabe sind neue Safety-Anforderungen in

textueller Form, die dieses Implementierungsmodell künftig einhalten soll, die es möglicherweise aber bisher verletzt. Die Ansätze erweitern das gegebene System dann um „Controller“, welche die Transitionen des Petrinetzes am Schalten hindern, wenn dadurch die Safety-Anforderungen verletzt würden. Obwohl diese Ansätze als „Synthese“ bezeichnet werden, wird nicht beschrieben, ob oder wie sie zur Erzeugung komplett neuer Systeme aus Spezifikationen eingesetzt werden können.

Insgesamt sind also die Konzepte zur Modellierung von lokalem Wissen ähnlich wie in dieser Arbeit, das gelöste Problem ist jedoch ein anderes. Auch die verwendeten Formalismen sind unterschiedlich: Während MSDs Anforderungen an nachrichtenbasierte Kommunikation beschreiben, kommunizieren die einzelnen Prozesse der Petrinetze nur implizit miteinander. Weiterhin wird die Synchronisation nicht direkt durch das Senden von Nachrichten durchgeführt, sondern über sogenannte Supervisor-Prozesse. Auch dabei wird die Kommunikation nicht explizit modelliert.

Ein ähnlicher Ansatz wird von Kalyon et al. [KLMM11] beschrieben. Dort wird ein gegebenes System aus asynchron kommunizierenden endlichen Automaten durch neu erzeugte lokale Controller so eingeschränkt, dass bestimmte Zustände nicht erreicht werden können. Dabei berechnen die lokalen Controller vorläufige Schätzungen der möglichen globalen Systemzustände. Auch in diesem Ansatz kommunizieren die Controller miteinander, um Informationen über den Gesamtzustand auszutauschen. Es ist jedoch auch mit diesem Ansatz nicht möglich, ein vollständig neues verteiltes System aus einer gegebenen Spezifikation zu erzeugen.

7.1.2 Ansätze ohne zusätzliche Synchronisation

Die folgenden Ansätze erzeugen nur dann eine korrekte Implementierung der gegebenen Spezifikation, wenn dazu die Nachrichten ausreichen, die bereits in dieser Spezifikation definiert sind. Andernfalls schlägt die Synthese entweder fehl oder sie resultiert in einer inkorrekten Implementierung, welche die Spezifikation nicht einhält.

Direkte Konstruktion von Controllern aus LSCs

Bontemps et al. [BHS05], sowie spätere Arbeiten von Harel et al. [HKP05], beschreiben Syntheseverfahren, die verteilte Implementierungen für gegebene LSC-Spezifikationen erzeugen. Dabei wird nur das explizit mittels LSCs spezifizierte Kommunikationsverhalten betrachtet – mit den oben erwähnten Nachteilen.

Bei Bontemps et al. werden zunächst für jedes Subsystem alle Lifelines der LSC-Spezifikation ermittelt, die dieses repräsentieren. Für jedes Subsystem wird dann ein Controller so erzeugt, dass er auf empfangene Nachrichten dieser Lifelines mit dem Senden einer Nachricht reagiert, die nach den Informationen dieses Subsystems in keinem LSC verboten, aber in mindestens einem LSC gefordert wird. Dabei wird die Kommunikation zwischen anderen Subsystemen jedoch ignoriert, wodurch Situationen auftreten können, in denen die Subsysteme Nachrichten senden müssen, die sie nach ihren lokalen Informationen nicht senden dürfen. Dies kann auch dann passieren, wenn die Spezifikation das Senden tatsächlich erlaubt. Die Controller senden die betreffenden Nachrichten in diesen Situationen nicht, wodurch Liveness-Anforderungen verletzt werden können.

Auch bei Harel et al. werden zunächst die Lifelines für die einzelnen Subsysteme in der LSC-Spezifikation ermittelt. Für jedes Subsystem wird dann ein Statechart gebildet, das je eine parallele Region für jede dieser Lifelines enthält. Das auf Basis der Lifelines gebildete Verhalten ist, ähnlich wie bei Bontemps et al., eine direkte Reaktion auf empfangene Nachrichten der jeweiligen Lifeline ohne Berücksichtigung anderer Kommunikation.

Beide Ansätze verwenden Model Checking, um festzustellen, ob die erzeugte Implementierung tatsächlich korrekt ist. Wie bereits erwähnt wurde, muss das jedoch nicht immer der Fall sein. Die Ansätze erzeugen dann keine alternative Implementierung, sondern melden lediglich ihren Fehlschlag. Daher sind sie *unvollständig* [BH05], d. h. sie finden nicht immer dann eine Implementierung, wenn eine solche der Spezifikation nach möglich ist.

Sun und Dong [SD05b] beschreiben ein Syntheseverfahren für LSCs, das eine verteilte Implementierung in Form eines *Communicating Sequential Processes (CSP)*-Systems erzeugt. Diese ist jedoch nur dann eine gültige Implementierung der gegebenen Spezifikation, wenn die Spezifikation bestimmten sehr restriktiven Annahmen genügt. Insbesondere muss sie bereits garantieren, dass die Aktionen des Systems niemals dazu führen können, dass eine Situation erreicht wird, in der das System die Liveness-Anforderungen nicht mehr ohne eine Verletzung der Safety-Anforderungen erfüllen kann. Solche „Sackgassen“ in einer Spezifikation sind aber durch die komplexen Wechselwirkungen der einzelnen LSCs bei größeren Spezifikationen schwer zu vermeiden. Sind diese dennoch in der Spezifikation vorhanden, so erkennt der Ansatz das nicht, sondern er erzeugt eine inkorrekte Implementierung, die die Anforderungen nicht immer erfüllt. Der in dieser Arbeit vorgestellte Ansatz andererseits funktioniert auch für Spezifikationen mit „Sackgassen“, da er Implementierungen, die eine solche erreichen würden, verwirft und Alternativen sucht.

Ähnliche Ansätze existieren für die Erzeugung verteilter Zustandsautomaten aus einer kombinierten LSC- und Z-Spezifikation [SD06], sowie für die Erzeugung von Verilog-Programmen aus LSCs [WQSD07]. Die Annahmen bei diesen Ansätzen sind ähnlich restriktiv wie bei dem oben erwähnten für die Erzeugung von CSP-Systemen.

Verteilung globaler Protokollautomaten ohne zusätzliche Nachrichten

Ein Ansatz von Halle und Bultan [HB10] erzeugt für ein gegebenes automatenbasiertes Kommunikationsprotokoll eines verteilten Systems lokale Sichten der Subsysteme auf den globalen Gesamtzustand. Dabei muss dieses Protokoll zunächst als ein einzelner Automat gegeben sein, also als globaler Controller. Der Ansatz modelliert die lokalen Informationen der Subsysteme über den aktuellen Zustand des globalen Protokolls auf ähnliche Weise wie das in dieser Arbeit vorgestellte Verfahren. Auf dieser Grundlage kann der Ansatz entscheiden, ob das Protokoll ohne zusätzliche Kommunikation implementierbar ist. In diesem Fall wird eine korrekte verteilte Implementierung des Protokolls erzeugt, d. h. korrekte lokale Controller.

Wenn das globale Protokoll jedoch ohne zusätzliche Nachrichten nicht verteilbar ist, dann meldet der Ansatz dies an den Anwender und erzeugt kein verteiltes Protokoll. Als mögliche Erweiterung wird vorgeschlagen, zusätzlich erforderliche Nachrichten automa-

tisch hinzuzufügen. Das Paper stellt jedoch keinen Algorithmus zur Umsetzung dieses Vorschlags vor und diskutiert auch nicht, wie eine solche Umsetzung erfolgen könnte.

Synthese ohne zusätzliche Kommunikation für andere Spezifikationen

Alur et al. [AMR+14] beschreiben die Erzeugung von Protokollen, die aus endlichen Automaten bestehen, für Spezifikationen auf Basis von *Message Sequence Charts (MSCs)*. Neben MSCs bestehen die Spezifikationen aus zusätzlich gegebenen Safety- und Liveness-Anforderungen, die beispielsweise als temporallogische Formeln formalisiert sein können. Weiterhin werden die Ein- und Ausgabealphabete der zu erzeugenden Prozesse und ein vollständiges Umgebungsmodell als Eingaben vorausgesetzt. Ähnlich wie bei dem in Kapitel 3 beschriebenen Ansatz werden zunächst Kandidaten für Controller-Automaten schrittweise erweitert, bis sie keine Deadlock-Zustände mehr enthalten und die Spezifikation erfüllen. Neben dem Umstand, dass das Verfahren keine zusätzliche Kommunikation in das System einfügt, ist ein weiterer Hauptunterschied die erste der beiden Phasen des Ansatzes: Aus den Szenarien wird zunächst für jeden Prozess ein „unvollständiger Automat“ abgeleitet, der bereits wesentliche Bestandteile des Prozessverhaltens definiert. Diese unvollständigen Automaten werden dann in der zweiten Phase des Ansatzes Gegenbeispiel-getrieben vervollständigt.

Finkbeiner und Schewe [FS05] betrachten das Problem der verteilten Synthese für Spezifikationen, die mittels ω-regulärer Sprachen definiert wurden. Ihr Ansatz berücksichtigt sowohl das beschränkte Wissen der zu synthetisierenden Prozesse als auch die Architektur des Systems. Laut den Autoren löst der Ansatz das Problem der verteilten Synthese für alle entscheidbaren Fälle. Dabei wird jedoch ein vollständig anderer Formalismus als in dieser Arbeit vorausgesetzt: Die Prozesse kommunizieren über gemeinsame Variablen und operieren vollständig synchron.

Eine Synthese ist für diese Art von Systemen den Autoren nach nur für Architekturen möglich, in denen kein *Information Fork* existiert. Dies ist eine Kommunikationsstruktur, in der einer der zu implementierenden Prozesse Informationen von der Umgebung erhält, die von einem anderen zu implementierenden Prozess nicht hergeleitet werden können. Wenn kein solcher Information Fork existiert, dann können die Prozesse nach ihrer Informiertheit sortiert und schrittweise als Automaten erzeugt werden.

Auch der Ansatz von Fridman und Puchala [FP11, Puc13] behandelt die Synthese von Prozessen aus Spezifikationen auf Basis ω-regulärer Sprachen. Die Synthese wird dazu als Spiel mit partieller Information der Spieler modelliert. Zu diesem Spiel wird eine Gewinnstrategie für die Prozesse ermittelt, aus der Implementierungen in Form endlicher Automaten abgeleitet werden. Wie bei Finkbeiner und Schewe ist der Formalismus sehr unterschiedlich zu dem in dieser Arbeit verwendeten.

Von Stefanescu et al. [SEM03, Ste06] wird ein Ansatz zur Erzeugung verteilter Automaten aus einem gegebenen globalen Transitionssystem beschrieben. Das Transitionssystem ist hierbei die Spezifikation; anders als bei LSCs oder MSDs muss also nicht erst ein globales Zustandsmodell gebildet werden, sondern es wird als Eingabe angenommen. Die Arbeit beschreibt die Synthese sowohl für synchron als auch für asynchron kommunizierende Automaten. Auch hier funktioniert der Ansatz nur, wenn das globale Transitionssystem ohne zusätzliche Nachrichten verteilbar ist.

7.1.3 Distributed LSCs

Als Alternative zu LSCs schlagen Fahland und Kantor [FK13] die *distributed LSCs (dLSCs)* vor. Jede Definition von Kommunikationsverhalten mittels dLSCs enthält alle zwischen den Subsystemen entstehenden Abhängigkeiten explizit. In dieser Spezifikationssprache ist es daher nicht möglich, dass implizite Abhängigkeiten zwischen Subsystemen entstehen. Dadurch wird der in LSCs (und MSDs) mögliche Fall ausgeschlossen, dass ein Subsystem auf die Kommunikation zweier anderer Subsysteme reagieren muss, da von dieser Kommunikation abhängt, ob es eine Nachricht senden muss oder nicht senden darf.

Durch die explizite Definition aller Abhängigkeiten in dLSCs sind niemals zusätzliche Synchronisationsnachrichten erforderlich, um die Spezifikation zu erfüllen. Der Nachteil ist jedoch, dass die Entwickler bereits bei der Erstellung der Spezifikation jede erforderliche Kommunikation manuell korrekt definieren müssen. Das in dieser Arbeit gelöste Problem wird also auf den Entwickler verlagert.

7.2 Zeitbehaftete Synthese

Ansätze zur Synthese zeitbehafteter Controller erzeugen diese meist als Timed Automata (siehe Abschnitt 2.2.2). Das gilt auch für die in dieser Arbeit entwickelte und in Kapitel 4 behandelte zeitbehaftete Erweiterung des Verfahrens aus Kapitel 3. Der in dieser Arbeit beschriebene Ansatz erzeugt jedoch, anders als die hier diskutierten verwandten Ansätze, *verteilte* Controller. Es gibt bisher offenbar keine anderen zeitbehafteten Syntheseverfahren, die verteilte Controller erzeugen. Die hier beschriebenen Ansätze sind jedoch in der Art verwandt, wie sie die zeitbehafteten Anteile der Controller berechnen.

Abschnitt 7.2.1 behandelt Ansätze, die – wie der zeitbehaftete Ansatz in Kapitel 4 dieser Arbeit – direkt auf Basis von Clock Zones operieren. Weitere Ansätze, die in Abschnitt 7.2.2 diskutiert werden, erzeugen aus den gegebenen Spezifikationen Eingabemodelle für andere Verfahren und nutzen diese zur Durchführung der eigentlichen Synthese. Abschnitt 7.2.3 schließlich befasst sich mit einem Template-basierten Ansatz zur zeitbehafteten Synthese.

7.2.1 Direkte Synthese auf Basis von Clock Zones

Diese Arbeit verwendet zur Berechnung der zeitbehafteten Anteile der erzeugten Controller ein Verfahren, das grob an dem von Maler et al. [MPS95] beschriebenen orientiert ist (siehe Abschnitt 4.3.2). Dieser Ansatz erzeugt einen globalen Controller in Form eines Timed Automaton durch direkte Berechnung der Clock Zones für das erlaubte Verhalten. Eine Kombination des Ansatzes mit einem nicht zeitbehafteten Ansatz von Liu und Smolka [LS98] wurde von Cassez et al. [CDF+05] beschrieben. Dieses Verfahren wurde in dem Werkzeug Uppaal Tiga implementiert, einer erweiterten Variante des Model Checkers Uppaal [BDL04].

Wie in dieser Arbeit (siehe Abschnitte 3.2.2 und 4.3.1) wird auch bei diesen Ansätzen das Syntheseproblem als ein Zwei-Spieler-Spiel des Systems gegen die Umgebung betrachtet, bei dem das System bei Erfüllung der Anforderungen oder Verletzung der

Umgebungsannahmen gewinnt. Der durch den jeweiligen Ansatz erzeugte globale Controller entspricht einer zeitbehafteten Gewinnstrategie für das System. Diese definiert alle (Re-)Aktionen des Systems für eine beliebige (zu den Annahmen konforme) Umgebung so, dass das System immer gewinnt.

Die beiden von Maler et al. und Cassez et al. beschriebenen Verfahren verwenden als Eingabe keine Variante von Sequenzdiagrammen, sondern ein System aus *Timed Game Automata (TGA)*. TGA sind eine Variante von Timed Automata, bei der zwischen zwei Arten von Transitionen unterschieden wird: Über das Schalten kontrollierbarer Transitionen entscheidet das System, während über das Schalten unkontrollierbarer Transitionen die Umgebung entscheidet. Das System aus TGA modelliert das Spiel, wobei zusätzliche temporallogische Formeln die Gewinnbedingung des Systems definieren. Die Verfahren erzeugen dann für das gegebene Spiel eine zeitbehaftete Gewinnstrategie für das System – sofern eine solche existiert. Diese entspricht einem korrekten globalen Controller.

7.2.2 Abbildung von LSCs/MSDs auf Timed Game Automata

Für die Synthese aus zeitbehafteten Sequenzdiagrammen gibt es zwei Ansätze, die die Sequenzdiagramme zunächst auf ein zeitbehaftetes Zwei-Spieler-Spiel abbilden und dann Uppaal Tiga zur Durchführung der eigentlichen Synthese verwenden. Das Spiel ist dabei jeweils durch ein System aus TGA modelliert. Indirekt kommt dabei also das in Abschnitt 7.2.1 diskutierte Verfahren von Cassez et al. zum Einsatz. Uppaal Tiga erzeugt eine zeitbehaftete Gewinnstrategie für das System, sofern eine solche für die gegebene Spezifikation existiert. Diese Gewinnstrategie entspricht dann einem globalen Controller.

Einer dieser beiden Ansätze ist das von Greenyer [Gre11] entwickelte zeitbehaftete Syntheseverfahren für MSDs. Da dieses Verfahren an demselben Lehrstuhl wie diese Arbeit entstand, wurde es bereits in Abschnitt 3.2.4 ausführlicher diskutiert. Das andere Verfahren wurde von Larsen et al. [LLNP10] beschrieben und basiert auf LSC-Spezifikationen. Hauptunterschied zwischen beiden Verfahren ist die explizite Modellierung von Umgebungsannahmen bei dem zuerst genannten. Beide Verfahren unterscheiden sich von der vorliegenden Arbeit – neben der erwähnten Beschränkung auf globale Controller – primär darin, dass sie nicht direkt auf Clock Zones operieren, sondern nur indirekt durch die Verwendung von Uppaal Tiga.

7.2.3 Template-basierte Synthese zeitbehafteter Controller

Finkbeiner et al. [FP12] beschreiben ein Verfahren zur Synthese zeitbehafteter Controller auf der Basis von *Templates*. Templates geben die Struktur eines Timed Automaton teilweise vor, sind aber durch Parameter anpassbar. Die Synthese eines Controllers auf Basis eines Templates besteht dann darin, die passenden Parameter für das Template zu ermitteln. Die Templates werden bereits vor Ausführung des Verfahrens manuell definiert. Der Vorteil dieses Ansatzes ist seine Effizienz. Der Hauptnachteil ist der erforderliche manuelle Aufwand bei der Definition der Templates. Zudem kann der Fall auftreten, dass mit dem gegebenen Template keine erfolgreiche Synthese möglich

ist. Das Verfahren liefert in diesem Fall keine Aussage darüber, ob der Fehlschlag der Synthese nur an einem zu speziellen Template liegt, oder daran, dass die Spezifikation widersprüchliche Anforderungen enthält.

7.3 Zusammenfassung

Es gibt einige Ansätze zur Synthese verteilter Controller, die zu dem in dieser Arbeit entwickelten Ansatz verwandt sind, da sie ebenfalls zu einer formalen Spezifikation automatisch ein verteiltes System erzeugen. Anders als die meisten dieser Ansätze fügt der hier beschriebene jedoch dem erzeugten System zusätzliche Kommunikation hinzu, wenn dies zur Erfüllung der Anforderungen erforderlich ist – und nur wenn die lokalen Informationen der Subsysteme nicht ausreichen. Von den verwandten Ansätzen führen nur wenige eine Synchronisation der Subsysteme abhängig von deren lokalen Informationen durch. Dies sind die von Katz, Peled und Schewe [KPS11, PS11, PS14] beschriebenen Ansätze und der von Kalyon et al. [KLMM11] beschriebene Ansatz. Diese Ansätze führen jedoch keine Synthese eines neuen verteilten Systems aus einer Spezifikation durch, sondern sie passen ein gegebenes System an neue Anforderungen an. Zudem betrachten sie keine MSD-Spezifikationen als Eingabe.

Grundsätzlich kann bei den übrigen verwandten Arbeiten im Bereich der verteilten Synthese aus Spezifikationen grob zwischen zwei Gruppen von Ansätzen unterschieden werden: Die Ansätze aus der einen Gruppe führen auch dann keine Nachrichten zusätzlich zu den in der Spezifikation definierten ein, wenn dies zur Implementierung erforderlich ist. Die Ansätze aus der anderen Gruppe führen in jedem Fall zusätzliche Kommunikation ein, auch wenn diese nicht erforderlich ist. Die Ergänzung zusätzlicher Kommunikation abhängig von der lokalen Informiertheit der Subsysteme ist daher ein Beitrag dieser Arbeit.

Die Ansätze zur Synthese zeitbehafteter Controller sind zu dem in dieser Arbeit beschriebenen Verfahren nur insofern verwandt, als sie ebenfalls das Teilproblem lösen, zeitbehaftetes Controllerverhalten zu erzeugen. Sie erzeugen jedoch keine verteilten Controller. Daher ist ein Beitrag dieser Arbeit die Kombination der Synthese zeitbehafteten Verhaltens mit dem neu vorgestellten verteilten Synthesealgorithmus (siehe Kapitel 4). Dazu gehört auch die Lösung der speziellen Probleme, die erst bei dieser Kombination auftreten.

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel schließt die Arbeit ab, indem es diese in Abschnitt 8.1 zusammenfasst und in Abschnitt 8.2 einen Ausblick auf mögliche zukünftige Arbeiten gibt.

8.1 Zusammenfassung

Diese Arbeit stellt einen Ansatz für die automatische Erzeugung von Modellen vor, welche das Verhalten jeweils eines Subsystems eines verteilten Gesamtsystems definieren. Diese automatische Erzeugung wird als Synthese bezeichnet. Voraussetzung für die Anwendung des Verfahrens ist, dass die Anforderungen an das Systemverhalten zunächst mittels *Modal Sequence Diagrams (MSDs)*, einer visuellen szenariobasierten Spezifikationssprache, modelliert wurden. Der Vorteil dieser Eingabesprache ist, dass sie die intuitive und gleichzeitig formale Spezifikation globaler Anforderungen an das Kommunikationsverhalten zwischen Subsystemen ermöglicht. Der entwickelte Syntheseansatz erzeugt die Verhaltensmodelle der Subsysteme so, dass das resultierende Gesamtsystem die Anforderungen der Spezifikation einhält, wenn die Umgebung des Systems die spezifizierten Annahmen einhält.

Das vorgestellte Syntheseverfahren ist prinzipiell allgemein für verteilte Systeme verwendbar. Diese Arbeit behandelt jedoch speziell die Anwendung für mechatronische Systeme, also Systeme mit mechanischen und elektronischen Anteilen, die von Software gesteuert werden. Diese Systeme weisen oft echtzeitkritisches Verhalten auf, also (Re-)Aktionen des Systems, die nicht zu früh oder zu spät erfolgen dürfen, da sonst schwere Unfälle auftreten können. Daher erweitert die Arbeit das entwickelte Syntheseverfahren so, dass die erzeugten Verhaltensmodelle auch zeitbehaftete Anforderungen einhalten, die in der gegebenen MSD-Spezifikation definiert sein können.

Die durch das vorgestellte Syntheseverfahren erzeugten Modelle enthalten im Idealfall alle plattformunabhängigen Informationen, die für eine automatische Codegenerierung benötigt werden. Ein solcher Idealfall liegt vor, wenn sich alle Anforderungen an das Systemverhalten vollständig mittels MSDs ausdrücken lassen. Während dies für nachrichtenbasierte Kommunikation der Fall ist, kann beispielsweise kontinuierliches Reglerverhalten durch MSDs nicht spezifiziert werden und die Modifikation von Variablen wird durch die Synthese nur eingeschränkt unterstützt. In vielen Fällen kann es daher erforderlich sein, die automatisch durch die Synthese erzeugten Modelle nachträglich anzupassen oder zu erweitern.

Um eine solche manuelle Weiterentwicklung zu unterstützen, behandelt diese Arbeit daher auch die Integration des Syntheseverfahrens in MechatronicUML, eine modellgetriebene Entwicklungsmethode speziell für mechatronische Systeme. Mechatronic-

UML definiert einen Entwicklungsprozess und eine UML-basierte Sprache, die Struktur- und Verhaltensmodelle beinhaltet. Diese Arbeit beschreibt daher die Übersetzung des Syntheseergebnisses in entsprechende MechatronicUML-Modelle und die Änderungen und Spezialfälle, die sich durch die Verwendung der Synthese im Entwicklungsprozess ergeben.

Zusammengefasst sind die drei Hauptbeiträge dieser Dissertation also die Entwicklung eines Ansatzes zur Synthese verteilter Systeme (siehe Kapitel 3), die Erweiterung dieses Syntheseansatzes für Echtzeitsysteme (siehe Kapitel 4) und seine Integration in den Entwicklungsansatz MechatronicUML (siehe Kapitel 5).

Insgesamt wurde im Rahmen dieser Arbeit also ein für verteilte mechatronische Systeme verwendbares Syntheseverfahren entwickelt, das die Entwicklung dieser Systeme durch die Erzeugung von (vorläufigen) Implementierungsmodellen unterstützt. Diese Modelle können dann in einem bestehenden modellgetriebenen Entwicklungsansatz weiterentwickelt werden. Das Syntheseverfahren ist dabei insbesondere für die Entwicklung sicherheitskritischer Systeme vorteilhaft, da die durch die Synthese erzeugten Modelle alle spezifizierten Anforderungen an das System einhalten.

Anders als bestehende Syntheseansätze für verteilte Systeme, modelliert der hier vorgestellte Ansatz das Wissen der Subsysteme über den Gesamtzustand des verteilten Systems und führt automatisch zusätzliche Kommunikation in das erzeugte System ein, wenn ein Subsystem für ein spezifikationskonformes Verhalten zusätzliches Wissen benötigt, über das andere Subsysteme verfügen. Andere Ansätze erzeugen entweder auch unnötige Kommunikation, oder sie sind für Spezifikationen, die zusätzliche Kommunikation benötigen, nicht anwendbar. Kapitel 7 behandelt die verwandten Arbeiten detaillierter.

Die in der Arbeit beschriebenen Konzepte wurden in Form von Plug-ins für die verbreitet eingesetzte Entwicklungsumgebung Eclipse implementiert. Diese Plug-ins basieren auf der durch Vorarbeiten entstandenen Werkzeugumgebung ScenarioTools und erweitern diese. ScenarioTools ermöglicht eine szenariobasierte Modellierung von Anforderungen mittels MSDs und deren Analyse durch Ansätze zur Simulation und Synthese. Die Implementierung wird in Kapitel 6 beschrieben, das zudem die im Rahmen der Arbeit durchgeführte Evaluierung diskutiert.

Zur Evaluierung des Syntheseverfahrens wurde seine Anwendung auf einige beispielhafte MSD-Spezifikationen untersucht, die sich an realen Systemen orientieren. In den untersuchten Fällen erzeugte das Verfahren jeweils korrekte Implementierungsmodelle, sofern die gegebene Spezifikation implementierbar war. Weiterhin wurden die Effizienz und Skalierbarkeit des Verfahrens mittels systematisch konstruierter MSD-Spezifikationen von variabler Größe untersucht.

Hierbei zeigte sich erwartungsgemäß, dass das Verfahren in der implementierten Form nur für relativ kleine Systeme mit vertretbarem Zeitaufwand durchgeführt werden kann. Die Komplexität von Systemen dieser Größe (bis ca. 40 einfache Controller) kann jedoch bei weitem ausreichen, um menschliche Entwickler zu überfordern – insbesondere, wenn das Ergebnis fehlerfrei sein muss. Das implementierte Verfahren kann oft innerhalb weniger Sekunden oder Minuten ein korrektes System implementieren, dessen manuelle Erstellung deutlich länger dauern würde. Zudem wäre für manuell erstellte Modelle zusätzlicher Verifikationsaufwand erforderlich. Dieser kann für ein automatisch synthe-

tisiertes System zumindest für alle in der MSD-Spezifikation erfassten Eigenschaften entfallen.

Das vorgestellte Syntheseverfahren lässt sich eingeschränkt auch in Fällen anwenden, in denen eine erfolgreiche Synthese durch eine zu komplexe Spezifikation nicht mit vertretbarem Zeitaufwand möglich ist: Die Anwendung der Synthese kann dann zwar nicht auf die gesamte Spezifikation, aber auf isolierte Teilbereiche, beispielsweise nur eine Teilmenge der Subsysteme, möglich sein. Eine Möglichkeit zur Verkleinerung einer gegebenen MSD-Spezifikation ist es, die Anforderungen abstrakter (mit weniger Nachrichten) zu modellieren und die Details, beispielsweise interne Nachrichten an Subkomponenten eines Subsystems, erst im Rahmen einer späteren Verfeinerung des Modells manuell zu ergänzen.

8.2 Ausblick

Diese Arbeit stellt einen neuen Ansatz zur Synthese verteilter Systeme vor, erweitert ihn für zeitbehaftete Systeme und behandelt seine Integration in eine bestehende Entwicklungsmethode. Bei der Lösung dieser drei Hauptaufgaben wurden jedoch immer wieder Probleme aufgedeckt, die zwar nicht in den Kern dieser Arbeit fallen, für eine praktische Anwendung des vorgestellten Verfahrens aber dennoch relevant sind. Da die Behandlung dieser Probleme den Rahmen dieser Arbeit sprengen würde, sind weitere zukünftige Arbeiten erforderlich, um sie zu lösen.

Da die in Kapitel 6 beschriebene Evaluierung des Ansatzes nicht anhand eines realen Systems erfolgte, verbleibt eine umfassendere Evaluierung im Rahmen industrieller Fallstudien als eine mögliche zukünftige Arbeit. Dadurch könnten Erkenntnisse über mögliche Probleme im praktischen Einsatz und dadurch erforderliche Anpassungen des Verfahrens gewonnen werden. Um den praktischen Nutzen des Verfahrens zu untersuchen, wären weiterhin empirische Studien für verschiedene Arten von Entwicklungsprojekten hilfreich. Im Rahmen dieser Arbeit wurde aufgrund des hohen Aufwands auf die Durchführung empirischer Studien verzichtet.

Bisher bieten sowohl das Syntheseverfahren als auch der Formalismus der MSDs keine Möglichkeit, kontinuierliches Verhalten zu berücksichtigen. Auch die direkte Modifikation von Speicherinhalten, beispielsweise Operationen auf Zeichenketten, werden nur sehr eingeschränkt unterstützt. Gelingt es in zukünftigen Arbeiten, Anforderungen an diese Verhaltensbestandteile bereits in der Synthese zu berücksichtigen, dann kann auf eine manuelle Weiterentwicklung in solchen Fällen verzichtet werden, in denen diese bislang zur Integration dieser Verhaltensbestandteile erforderlich ist. Zunächst muss dazu aber eine Erweiterung des Formalismus der MSDs erfolgen, um beispielsweise auf kontinuierliches Verhalten Bezug nehmen zu können.

In dieser Arbeit wurde zunächst ein grundlegender Prototyp des Syntheseverfahrens umgesetzt, ohne besonderes Augenmerk auf mögliche Optimierungen zu setzen. Um die praktische Anwendbarkeit des Verfahrens für größere bzw. komplexere Systeme zu verbessern, könnten zukünftige Arbeiten das Ziel einer Verringerung von Laufzeit und Speicherbedarf des Verfahrens anstreben. Beispielsweise könnte eine Parallelisierung des Algorithmus die Laufzeit des Verfahrens auf aktuellen Multi-Core- und zukünftigen Ma-

ny-Core-Systemen verbessern. Eine naheliegende Möglichkeit zur Parallelisierung wäre beispielsweise die gleichzeitige Konstruktion mehrerer alternativer Controller-Kandidaten, da zwischen diesen keine Abhängigkeiten bestehen.

Im Fall, dass mehr als eine korrekte Implementierung für eine MSD-Spezifikation möglich ist, erzeugt der beschriebene Synthesealgorithmus aktuell nur eine beliebige dieser Varianten. Wurde eine Implementierung gefunden, so bricht der Algorithmus ab, ohne nach Alternativen zu suchen. Dies ist einerseits ein Vorteil, weil Laufzeit und Speicherbedarf des Algorithmus dadurch geringer bleiben. Andererseits kann es aber sein, dass eine Implementierung mit bestimmten Eigenschaften gewünscht ist, beispielsweise eine Implementierung, deren Controller maximal eine bestimmte Anzahl an Zuständen haben. Während diese Eigenschaft durch Verwerfen von gefundenen Implementierungen ohne die Eigenschaft erfüllt werden kann, ist in anderen Fällen das Betrachten *aller* Alternativen erforderlich: Dies ist der Fall, wenn gefordert wird, dass die Implementierung in einer bestimmten Eigenschaft, etwa der Anzahl erforderlicher Synchronisationsnachrichten, minimal oder maximal ist. Es übersteigt den Umfang dieser Arbeit, derartige zusätzliche Anforderungen an die Implementierung zu unterstützen.

Wie in Abschnitt 8.1 erwähnt wurde, kann es sinnvoll sein, das Syntheseverfahren aus Effizienzgründen nur auf Teile der Spezifikation anzuwenden. Dabei muss die Aufteilung so erfolgen, dass alle resultierenden Spezifikationen unabhängig voneinander implementiert werden können. Die Implementierungen aller diese Spezifikationen müssen zudem zusammengenommen die ursprüngliche Spezifikation erfüllen. Zur manuellen Aufteilung einer gegebenen MSD-Spezifikation existiert bereits ein Verfahren [Gre11, GK13]. Die Anforderungen werden dabei abhängig davon aufgeteilt, auf welche Subsysteme diese sich beziehen. Um die dadurch entstehenden neuen Spezifikationen unabhängig voneinander implementieren zu können, werden diesen manuell jeweils zusätzliche Annahmen über das Verhalten der übrigen Subsysteme hinzugefügt. Den Spezifikationen dieser anderen Subsysteme werden dieselben Annahmen (ebenfalls manuell) als Anforderungen hinzugefügt. Das mögliche Verhalten der Subsysteme wird dadurch bereits vor der Synthese so weit konkretisiert, dass die Implementierungen der Subsysteme nicht inkompatibel zueinander sein können. Die Anwendung dieser Technik wurde jedoch noch nicht für verteilte Systeme untersucht. Zukünftige Arbeiten könnten den Ansatz entsprechend anpassen.

Wie bereits in Abschnitt 5.2 diskutiert wurde, wäre eine Werkzeugunterstützung für den Fall hilfreich, dass eine MSD-Spezifikation sich nach manuellen Änderungen an den erzeugten Controllern ebenfalls ändert. Dann sollten die manuellen Änderungen, soweit möglich, in einer erneuten Synthese berücksichtigt werden. Ein anderer problematischer Fall sind manuelle Änderungen an den erzeugten Controllern, welche diese so weit verändern, dass sie einer anderen korrekten Implementierung entsprechen. Um letzteres automatisiert nachweisen zu können, wäre ein Model-Checking-Ansatz erforderlich, der ein gegebenes Controllersystem auf Einhaltung einer gegebenen MSD-Spezifikation überprüft. Ein solches Verfahren könnte basierend auf dem Syntheseverfahren entwickelt werden, da dieses als Zwischenschritt der Synthese bereits einen gegebenen Controllersystem-Kandidaten darauf untersuchen kann, ob dieser eine Lösung ist.

Literaturverzeichnis

[ABD+14] Anacker, Harald; Brenner, Christian; Dorociak, Rafal; Dumitrescu, Roman; Gausemeier, Jürgen; Iwanek, Peter; Schäfer, Wilhelm; Vassholz, Mareen: Methods for the Domain-Spanning Conceptual Design. In: *Design Methodology for Intelligent Technical Systems Systems – Develop Intelligent Technical Systems of the Future*. Springer, Januar 2014, Kapitel 4, S. 119–185

[ACD90] Alur, Rajeev; Courcoubetis, Costas; Dill, David L.: Model-Checking for Real-Time Systems. In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, 1990, S. 414–425

[ACD93] Alur, Rajeev; Courcoubetis, Costas; Dill, David L.: Model-checking in Dense Real-time. In: *Information and Computation* 104 (1993), Mai, Nr. 1, S. 2–34. – ISSN 0890–5401

[AD94] Alur, Rajeev; Dill, David L.: A Theory of Timed Automata. In: *Theoretical Computer Science* 126 (1994), April, Nr. 2, S. 183–235. – ISSN 0304–3975

[AMR+14] Alur, Rajeev; Martin, Milo; Raghothaman, Mukund; Stergiou, Christos; Tripakis, Stavros; Udupa, Abhishek: Synthesizing Finite-State Protocols from Scenarios and Requirements. In: Yahav, Eran (Hrsg.): *Hardware and Software: Verification and Testing* Bd. 8855. Springer, 2014. – ISBN 978–3–319–13337–9, S. 75–91

[BB91] Benveniste, Albert; Berry, Gérard: The Synchronous Approach to Reactive and Real-time Systems. In: *Another Look at Real-Time Programming, Proceedings of the IEEE* 79 (1991), September, Nr. 9, S. 1270–1282. – ISSN 0018–9219

[BBB+12] Becker, Steffen; Brenner, Christian; Brink, Christopher; Dziwok, Stefan; Löffler, Renate; Heinzemann, Christian; Pohlmann, Uwe; Schäfer, Wilhelm; Suck, Julian; Sudmann, Oliver: The MechatronicUML Design Method – Process, Syntax, and Semantics / Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn. 2012 (tr-ri-12-326). – Forschungsbericht. Version 0.3

[BBD+12] Becker, Steffen; Brenner, Christian; Dziwok, Stefan; Gewering, Thomas; Heinzemann, Christian; Pohlmann, Uwe; Priesterjahn, Claudia; Schäfer, Wilhelm; Suck, Julian; Sudmann, Oliver; Tichy, Matthias:

The MechatronicUML Method – Process, Syntax, and Semantics / Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn. 2012 (tr-ri-12-318). – Forschungsbericht. Version 0.2

[Büc90] Büchi, Julius R.: On a Decision Method in Restricted Second Order Arithmetic. In: Mac Lane, Saunders (Hrsg.); Siefkes, Dirk (Hrsg.): *The Collected Works of J. Richard Büchi*. Springer, 1990. – ISBN 978–1–4613–8930–9, S. 425–435

[BDG+14a] Becker, Steffen; Dziwok, Stefan; Gerking, Christopher; Heinzemann, Christian; Schäfer, Wilhelm; Meyer, Matthias; Pohlmann, Uwe: The MechatronicUML Method: Model-Driven Software Engineering of Self-Adaptive Mechatronic Systems. In: *Proceedings of the 36th International Conference on Software Engineering (Posters)*, ACM, New York, NY, USA, Mai 2014

[BDG+14b] Becker, Steffen; Dziwok, Stefan; Gerking, Christopher; Schäfer, Wilhelm; Heinzemann, Christian; Thiele, Sebastian; Meyer, Matthias; Priesterjahn, Claudia; Pohlmann, Uwe; Tichy, Matthias: The MechatronicUML Design Method - Process and Language for Platform-Independent Modeling / Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn. 2014 (tr-ri-14-337). – Forschungsbericht. Version 0.4

[BDL04] Behrmann, Gerd; David, Alexandre; Larsen, Kim G.: A Tutorial on Uppaal. In: Bernardo, Marco (Hrsg.); Corradini, Flavio (Hrsg.): *Formal Methods for the Design of Real-Time Systems* Bd. 3185. Springer, 2004. – ISBN 978–3–540–23068–7, S. 200–236

[BFG+03] Burmester, Sven; Flake, Stephan; Giese, Holger; Schäfer, Wilhelm; Tichy, Matthias: Towards the Compositional Verification of Real-Time UML Designs. In: *Proceedings of the 9th European Software Engineering Conference (ESEC 2003)*. Helsinki, Finland : ACM Press, New York, NY, USA, September 2003, S. 38–47

[BGH+14] Brenner, Christian; Greenyer, Joel; Holtmann, Jörg; Liebel, Grischa; Stieglbauer, Gerald; Tichy, Matthias: ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study. In: *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)* Bd. 67, EASST, 2014

[BGP13] Brenner, Christian; Greenyer, Joel; Panzica La Manna, Valerio: The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions. In: *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)* Bd. 58, EASST, 2013

[BGS05] Burmester, Sven; Giese, Holger; Schäfer, Wilhelm: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: Hartman, Alan (Hrsg.); Kreische, David (Hrsg.): *Model Driven Architecture - Foundations and Applications* Bd. 3748. Springer, 2005. – ISBN 978–3–540–30026–7, S. 25–40

[BGS15] Brenner, Christian; Greenyer, Joel; Schäfer, Wilhelm: On-the-fly Synthesis of Scarcely Synchronizing Distributed Controllers from Scenario-Based Specifications. In: Egyed, Alexander (Hrsg.); Schaefer, Ina (Hrsg.): *Fundamental Approaches to Software Engineering (FASE 2015)* Bd. 9033. Springer, 2015. – ISBN 978–3–662–46674–2, S. 51–65

[BH05] Bontemps, Yves; Heymans, Patrick: From Live Sequence Charts to State Machines and Back: A Guided Tour. In: *IEEE Transactions on Software Engineering* 31 (2005), Nr. 12, S. 999–1014

[BHS05] Bontemps, Yves; Heymans, Patrick; Schobbens, Pierre-Yves: Lightweight Formal Methods for Scenario-Based Software Engineering. In: Leue, Stefan (Hrsg.); Systä, Tarja (Hrsg.): *Scenarios: Models, Transformations and Tools* Bd. 3466. Springer, 2005. – ISBN 978–3–540–26189–6, S. 174–192

[BHSH13] Brenner, Christian; Heinzemann, Christian; Schäfer, Wilhelm; Henkler, Stefan: Automata-Based Refinement Checking for Real-Time Systems. In: *Proceedings of Software Engineering 2013 – Fachtagung des GI-Fachbereichs Softwaretechnik*, Gesellschaft für Informatik, 26 Februar - 1 März 2013, S. 99–112

[BK07] Baier, Christel; Katoen, Joost-Pieter: *Principles of Model Checking*. 1. MIT Press, 2007. – ISBN 978–0–262–02649–9

[BS03] Bontemps, Yves; Schobbens, Pierre-Yves: Synthesis of Open Reactive Systems from Scenario-Based Specifications. In: *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD 2003)*, 2003, S. 41–50

[BSL04] Bontemps, Yves; Schobbens, Pierre-Yves; Löding, Christof: Synthesis of Open Reactive Systems from Scenario-Based Specifications. In: *Fundamenta Informaticae* 62 (2004), Februar, Nr. 2, S. 139–169. – ISSN 0169–2968

[BY04] Bengtsson, Johan; Yi, Wang: Timed Automata: Semantics, Algorithms and Tools. In: Desel, Jörg (Hrsg.); Reisig, Wolfgang (Hrsg.); Rozenberg, Grzegorz (Hrsg.): *Lectures on Concurrency and Petri Nets* Bd. 3098. Springer, 2004. – ISBN 978–3–540–22261–3, S. 87–124

[CDF+05] Cassez, Franck; David, Alexandre; Fleury, Emmanuel; Larsen, Kim G.; Lime, Didier: Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In: Abadi, Martín (Hrsg.); de Alfaro, Luca (Hrsg.): *CONCUR 2005 - Concurrency Theory* Bd. 3653. Springer, 2005. – ISBN 978–3–540–28309–6, S. 66–80

[DBHT12] Dziwok, Stefan; Bröker, Kathrin; Heinzemann, Christian; Tichy, Matthias: A Catalog of Real-Time Coordination Patterns for Advanced Mechatronic Systems / Universität Paderborn, Deutschland. 2012 (tr-ri-12-319). – Forschungsbericht

[DGB+14] Dziwok, Stefan; Gerking, Christopher; Becker, Steffen; Thiele, Sebastian; Heinzemann, Christian; Pohlmann, Uwe: A Tool Suite for the Model-Driven Software Engineering of Cyber-Physical Systems. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, S. 715–718

[DH98] Damm, Werner; Harel, David: LSCs: Breathing Life into Message Sequence Charts / Weizmann Institute of Science, Faculty of Mathematics and Computer Science. 1998 (CS98-09). – Forschungsbericht

[DH01] Damm, Werner; Harel, David: LSCs: Breathing Life into Message Sequence Charts. In: *Formal Methods in System Design* Bd. 19, Kluwer Academic Publishers, 2001, S. 45–80

[DHT12] Dziwok, Stefan; Heinzemann, Christian; Tichy, Matthias: Real-Time Coordination Patterns for Advanced Mechatronic Systems. In: Sirjani, Marjan (Hrsg.): *Coordination Models and Languages* Bd. 7274. Springer, 2012. – ISBN 978–3–642–30828–4, S. 166–180

[Dil90] Dill, David L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Springer, 1990. – ISBN 0–387–52148–8, S. 197–212

[DM01] David, Alexandre; Möller, M. Oliver: From HUppaal to Uppaal: A Translation from Hierarchical Timed Automata to Flat Timed Automata / BRICS. 2001 (RS-01-11). – Forschungsbericht

[DP14] Dann, Andreas P.; Pohlmann, Uwe: The MechatronicUML Hardware Platform Description Method – Process and Language / Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn. 2014 (tr-ri-14-336). – Forschungsbericht. Version 0.1

[EH11] Eckardt, Tobias; Heinzemann, Christian: Providing Timing Computations for FUJABA. In: *Proceedings of the 8th International Fujaba Days (University of Tartu, Estonia)*, 2011

[EHH+13] Eckardt, Tobias; Heinzemann, Christian; Henkler, Stefan; Hirsch, Martin; Priesterjahn, Claudia; Schäfer, Wilhelm: Modeling and verifying dynamic communication structures based on graph transformations. In: *Computer Science - Research and Development* 28 (2013), Februar, Nr. 1, S. 3–22

[FG12] Frieben, Jens; Greenyer, Joel: Consistency Checking Scenario-Based Specifications of Dynamic Systems. In: *Proceedings of the 4th Workshop on Behavioural Modelling – Foundations and Application (BM-FA 2012)*, ACM, 2012. – ISBN 978–1–4503–1187–8

[FK13] Fahland, Dirk; Kantor, Amir: Synthesizing Decentralized Components from a Variant of Live Sequence Charts. In: *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013)*, 2013, S. 25–38

[FP11] Fridman, Wladimir; Puchala, Bernd: Distributed Synthesis for Regular and Contextfree Specifications. In: Murlak, Filip (Hrsg.); Sankowski, Piotr (Hrsg.): *Mathematical Foundations of Computer Science 2011* Bd. 6907. Springer, 2011. – ISBN 978–3–642–22992–3, S. 532–543

[FP12] Finkbeiner, Bernd; Peter, Hans-Jörg: Template-Based Controller Synthesis for Timed Systems. In: Flanagan, Cormac (Hrsg.); König, Barbara (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems* Bd. 7214. Springer, 2012. – ISBN 978–3–642–28755–8, S. 392–406

[Fra06] Frank, Ursula: *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme*, Fakultät für Maschinenbau, Universität Paderborn, Dissertation, 2006

[FS05] Finkbeiner, Bernd; Schewe, Sven: Uniform Distributed Synthesis. In: *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, 2005, S. 321–330

[GB03] Giese, Holger; Burmester, Sven: Real-Time Statechart Semantics / Lehrstuhl für Softwaretechnik, Universität Paderborn. 2003 (tr-ri-03-239). – Forschungsbericht

[GB04] Giese, Holger; Burmester, Sven: Analysis and Synthesis for Parameterized Timed Sequence Diagrams. In: *Proceedings of the 3rd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (ICSE 2003 Workshop W5S), Edinburgh, Schottland*, 2004

[GBC+13] Greenyer, Joel; Brenner, Christian; Cordy, Maxime; Heymans, Patrick; Gressi, Erika: Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, ACM, 2013, S. 433–443

[GFDK09] Gausemeier, Jürgen; Frank, Ursula; Donoth, Jörg; Kahl, Sascha: Specification Technique for the Description of Self-Optimizing Mechatronic Systems. In: *Research in Engineering Design* 20 (2009), Nr. 4, S. 201–223. – ISSN 0934–9839

[Gie03] Giese, Holger: Towards Scenario-Based Synthesis for Parametric Timed Automata. In: *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM, ICSE 2003 Workshop 8), Portland, USA*, 2003

[GK13] Greenyer, Joel; Kindler, Ekkart: Compositional Synthesis of Controllers from Scenario-Based Assume-Guarantee Specifications. In: Moreira, Ana (Hrsg.); Schätz, Bernhard (Hrsg.); Gray, Jeff (Hrsg.); Vallecillo, Antonio (Hrsg.); Clarke, Peter (Hrsg.): *Model-Driven Engineering Languages and Systems: Proceedings of the ACM/IEEE 16th International Conference (MODELS 2013)* Bd. 8107. Springer, 2013. – ISBN 978–3–642–41532–6, S. 774–789

[GPBG13] Greenyer, Joel; Panzica La Manna, Valerio; Brenner, Christian; Ghezzi, Carlo: Synthesizing Safe Dynamic Updates from Evolving Specifications / Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano. 2013 (n. 2013.2). – Forschungsbericht

[Gre11] Greenyer, Joel: *Scenario-based Design of Mechatronic Systems*, Universität Paderborn, Dissertation, 2011

[GRS14] Gausemeier, Jürgen (Hrsg.); Rammig, Franz-Josef (Hrsg.); Schäfer, Wilhelm (Hrsg.): *Design Methodology for Intelligent Technical Systems - Develop Intelligent Technical Systems of the Future*. Springer, Januar 2014. – ISBN 978–3–642–45434–9

[GRSS11] Greenyer, Joel; Rieke, Jan; Schäfer, Wilhelm; Sudmann, Oliver: The Mechatronic UML Development Process. In: Tarr, Peri L. (Hrsg.); Wolf, Alexander L. (Hrsg.): *Engineering of Software*. Springer, 2011. – ISBN 978–3–642–19822–9, S. 311–322

[GS13] Giese, Holger; Schäfer, Wilhelm: Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML. In: Cámara, Javier (Hrsg.); de Lemos, Rogerio (Hrsg.); Ghezzi, Carlo (Hrsg.); Lopes, Antonia (Hrsg.): *Assurances for Self-Adaptive Systems* Bd. 7740. Springer, 2013, S. 152–186

[GT05] Giese, Holger; Tissen, Sergej: The SceBaSy PlugIn for the Scenario-Based Synthesis of Real-Time Coordination Patterns for Mechatronic UML. In: *Proceedings of the 3rd International Fujaba Days*, 2005, S. 67–70

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Juni, Nr. 3, S. 231–274. – ISSN 0167–6423

[HB10] Halle, Sylvain; Bultan, Tevfik: Realizability Analysis for Message-based Interactions Using Shared-State Projections. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2010)*, 2010

[HB13] Heinzemann, Christian; Becker, Steffen: Executing Reconfigurations in Hierarchical Component Architectures. In: *Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering*, ACM, Juni 2013

[HBDS15] Heinzemann, Christian; Brenner, Christian; Dziwok, Stefan; Schäfer, Wilhelm: Automata-based refinement checking for real-time systems. In: *Computer Science - Research and Development* 30 (2015), Juni, Nr. 3-4, S. 255–283. – Online veröffentlicht im Juni 2014

[HK02] Harel, David; Kugler, Hillel: Synthesizing State-Based Object Systems from LSC Specifications. In: *Foundations of Computer Science* 13:1 (2002), S. 5–51

[HKP05] Harel, David; Kugler, Hillel; Pnueli, Amir: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: *Formal Methods in Software and Systems Modeling* Bd. 3393, Springer, 2005, S. 309–324

[HM02a] Harel, David; Marelly, Rami: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. In: *Software and System Modeling (SoSyM)* 2 (2002), S. 2003

[HM02b] Harel, David; Marelly, Ramy: Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In: *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02), Fort Worth, Texas*, 2002, S. 193–202

[HM03] Harel, David; Marelly, Rami: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. 1. Auflage. Springer, 2003

[HM08] Harel, David; Maoz, Shahar: Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In: *Software and Systems Modeling (SoSyM)* 7 (2008), Nr. 2, S. 237–252

[HRS13] Heinzemann, Christian; Rieke, Jan; Schäfer, Wilhelm: Simulating Self-Adaptive Component-Based Systems using MATLAB/Simulink. In: *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '13)*, IEEE Computer Society Press, September 2013, S. 71–80

[HS14] Holtmann, Jörg; Shipchanov, Dimitar: Considering Architectural Properties in Real-time Play-out. In: *Proceedings of 12th Workshop Automotive Software Engineering* Bd. P-232, Köllen, September 2014, S. 2169–2180

[HSD+15] Heinzemann, Christian; Schubert, David; Dziwok, Stefan; Pohlmann, Uwe; Priesterjahn, Claudia; Brenner, Christian; Schäfer, Wilhelm:

RailCab Convoys: An Exemplar for Using Self-Adaptation in Cyber-Physical Systems / Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn. 2015 (tr-ri-15-344). – Forschungsbericht

[HSST13] Heinzemann, Christian; Sudmann, Oliver; Schäfer, Wilhelm; Tichy, Matthias: A Discipline-Spanning Development Process for Self-Adaptive Mechatronic Systems. In: *Proceedings of the 2013 International Conference on Software and System Process*, ACM, 18 - 19 Mai 2013 (ICSSP 2013), S. 36–45

[Int11] International Telecommunication Union: Recommendation Z.120: Message Sequence Chart (MSC). (2011)

[KHD14] Koch, Thorsten; Holtmann, Jörg; DeAntoni, Julien: Generating EAST-ADL Event Chains from Scenario-based Requirements Specifications. In: Avgeriou, Paris (Hrsg.); Zdun, Uwe (Hrsg.): *Proceedings of the 8th European Conference on Software Architecture (ECSA 2014)* Bd. 8627, Springer, August 2014, S. 146–153

[KLMM11] Kalyon, Gabriel; Le Gall, Tristan; Marchand, Hervé; Massart, Thierry: Synthesis of Communicating Controllers for Distributed Systems. In: *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC 2011)*, 2011. – ISSN 0743–1546, S. 1803–1810

[KPS11] Katz, Gal; Peled, Doron; Schewe, Sven: Synthesis of Distributed Control through Knowledge Accumulation. In: Gopalakrishnan, Ganesh (Hrsg.); Qadeer, Shaz (Hrsg.): *Computer Aided Verification* Bd. 6806. Springer, 2011. – ISBN 978–3–642–22109–5, S. 510–525

[KTWW06] Klose, Jochen; Toben, Tobe; Westphal, Bernd; Wittke, Hartmut: Check It Out: On the Efficient Formal Verification of Live Sequence Charts. In: Ball, Thomas (Hrsg.); Jones, Robert B. (Hrsg.): *Computer Aided Verification* Bd. 4144. Springer, 2006. – ISBN 978–3–540–37406–0, S. 219–233

[LL95] Lewerentz, Claus; Lindner, Thomas: "Production Cell": A Comparative Study in Formal Specification and Verification. In: *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, Springer, 1995. – ISBN 3–540–60589–4, S. 388–416

[LLNP10] Larsen, Kim G.; Li, Shuhao; Nielsen, Brian; Pusinskas, Saulius: Scenario-based Analysis and Synthesis of Real-time Systems Using UPPAAL. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010)*, European Design and Automation Association, 2010. – ISBN 978–3–9810801–6–2, S. 447–452

[LS98] Liu, Xinxin; Smolka, Scott A.: Simple Linear-Time Algorithms for Minimal Fixed Points. In: Larsen, Kim G. (Hrsg.); Skyum, Sven (Hrsg.);

Winskel, Glynn (Hrsg.): *Automata, Languages and Programming* Bd. 1443. Springer, 1998. – ISBN 978–3–540–64781–2, S. 53–66

[MHK02] Marelly, Rami; Harel, David; Kugler, Hillel: Multiple Instances and Symbolic Variables in Executable Sequence Charts. In: *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)* Bd. 37, 2002, S. 83–100

[MPS95] Maler, Oded; Pnueli, Amir; Sifakis, Joseph: On the Synthesis of Discrete Controllers for Timed Systems. In: Mayr, Ernst W. (Hrsg.); Puech, Claude (Hrsg.): *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1995)* Bd. 900. Springer, 1995. – ISBN 978–3–540–59042–2, S. 229–242

[Obj10] Object Management Group (OMG). *Object Constraint Language (OCL 2.2)*. Februar 2010

[Obj11] Object Management Group (OMG). *UML 2.4.1 Superstructure Specification*. August 2011

[PDM+14] Pohlmann, Uwe; Dziwok, Stefan; Meyer, Matthias; Tichy, Matthias; Thiele, Sebastian: A Modelica Coordination Pattern Library for Cyber-Physical Systems. In: *Proceedings of the 7th International Conference on Simulation Tools and Techniques (ICST 2014)*, 2014

[PGGB13] Panzica La Manna, Valerio; Greenyer, Joel; Ghezzi, Carlo; Brenner, Christian: Formalizing Correctness Criteria of Dynamic Updates Derived from Specification Changes. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)*, 2013

[PS11] Peled, Doron; Schewe, Sven: Practical Distributed Control Synthesis. In: Yu, Fang (Hrsg.); Wang, Chao (Hrsg.): *Proceedings of the 13th International Workshop on Verification of Infinite-State Systems (INFINITY 2011)* Bd. 73, 2011, S. 2–17

[PS14] Peled, Doron; Schewe, Sven: Distributed Control Synthesis. In: Voronkov, Andrei (Hrsg.); Korovina, Margarita (Hrsg.): *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, EasyChair, 2014, S. 271–288

[Puc13] Puchala, Bernd: *Synthesis of Winning Strategies for Interaction under Partial Information*, RWTH Aachen, Dissertation, 2013

[RDS+12] Rieke, Jan; Dorociak, Rafal; Sudmann, Oliver; Gausemeier, Jürgen; Schäfer, Wilhelm: Management of Cross-Domain Model Consistency for Behavior Models of Mechatronic Systems. In: *Proceedings of the International Design Conference (DESIGN 2012)*, 2012

[SD05a] Sun, Jun; Dong, Jin S.: Model Checking Live Sequence Charts. In: *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, 2005, S. 529–538

[SD05b] Sun, Jun; Dong, Jin S.: Synthesis of Distributed Processes from Scenario-based Specifications. In: *Proceedings of the International Conference on Formal Methods (FM 2005)*, Springer, 2005. – ISBN 3–540–27882–6, 978–3–540–27882–5, S. 415–431

[SD06] Sun, Jun; Dong, Jin S.: Design Synthesis from Interaction and State-based Specifications. In: *IEEE Transactions on Software Engineering* 32 (2006), Juni, Nr. 6, S. 349–364. – ISSN 0098–5589

[SEM03] Stefanescu, Alin; Esparza, Javier; Muscholl, Anca: Synthesis of Distributed Algorithms Using Asynchronous Automata. In: Amadio, Roberto (Hrsg.); Lugiez, Denis (Hrsg.): *CONCUR 2003 - Concurrency Theory* Bd. 2761. Springer, 2003. – ISBN 978–3–540–40753–9, S. 27–41

[Ste06] Stefanescu, Alin: *Automatic Synthesis of Distributed Transition Systems*, Universität Stuttgart, Dissertation, 2006

[TA99] Tripakis, Stavros; Altisen, Karine: On-the-Fly Controller Synthesis for Discrete and Dense-Time Systems. In: Wing, Jeannette M. (Hrsg.); Woodcock, Jim (Hrsg.); Davies, Jim (Hrsg.): *FM'99 Formal Methods* Bd. 1708. Springer, 1999. – ISBN 978–3–540–66587–8, S. 233–252

[VDI04] VDI, Verein Deutscher Ingenieure: *Richtlinie VDI 2206, Entwicklungsmethodik für mechatronische Systeme*. Berlin: VDI, Verein Deutscher Ingenieure, 2004

[WGT14] Winetzhammer, Sabine; Greenyer, Joel; Tichy, Matthias: Integrating Graph Transformations and Modal Sequence Diagrams for Specifying Structurally Dynamic Reactive Systems. In: Amyot, Daniel (Hrsg.); Fonseca i Casas, Pau (Hrsg.); Mussbacher, Gunter (Hrsg.): *System Analysis and Modeling: Models and Reusability (SAM 2014)* Bd. 8769. Springer, 2014. – ISBN 978–3–319–11742–3, S. 126–141

[WL97] Weise, Carsten; Lenzkes, Dirk: Efficient Scaling-Invariant Checking of Timed Bisimulation. In: *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS 97), LNCS 1200*, Springer, 1997, S. 177–188

[WQSD07] Wang, Hai H.; Qin, Shengchao; Sun, Jun; Dong, Jin S.: Realizing Live Sequence Charts in SystemVerilog. In: *Proceedings of the 6th International Symposium on Theoretical Aspects of Software Engineering* (2007), S. 379–388. ISBN 0–7695–2856–2

Abbildungsverzeichnis

Index