Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Erweiterte Simulationsumgebung für ein verbessertes Verständnis von Widersprüchen in szenariobarierten Spezifikationen

Bachelorarbeit

im Studiengang Informatik

von

Rebecca Lühmann

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 10.03.2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 10.03.2015

Rebecca Lühmann

Zusammenfassung

Angenommen Sie sind Ingenieur und haben ein komplexes System spezifiziert, welches ständig auf verschiedene Umweltereignisse reagieren soll. Sie haben dafür viele Szenarien erstellt, welche nun von einem Programm untersucht werden. Dabei findet es heraus, dass es Abfolgen von Ereignissen gibt, auf die Ihr System nicht reagieren kann, ohne dass es eine Ihrer spezifizierten Anforderungen verletzt. Aber wie finden Sie heraus, was Sie an Ihrer Spezifizierung korrigieren müssen?

Diese Bachelorarbeit befasst sich mit der Frage, wie Widersprüche einer szenariobasierten Spezifikation computergestützt aufbereitet werden können, sodass sie schneller behoben werden können. Es gibt bereits die Möglichkeit, in so genannten Play-Outs Schritt für Schritt das spezifizierte System bis zum unerwünschten Verhalten zu verfolgen. Jedoch fehlt die Übersicht, um verschiedene Zustände gezielt zu untersuchen, in denen unerwünschtes Verhalten auftritt. Hinsichtlich dieser Übersichtlichkeit werden Konzepte vorgestellt. Diese Konzepte informieren Sie als Ingenieur über verschiedene relevante Eigenschaften des aktuellen und der erreichbaren unerwünschten Zustände. Es werden zudem aussagekräftige Vergleiche der unterschiedlichen Fehlerzustände angeboten. Am Ende liegt es zwar wieder am Anwender, zu prüfen, ob eine mögliche Korrektur im Modell auch realistisch ist. Nun wird ihm bis zu diesem Punkt allerdings viel Arbeit abgenommen. Eine Bachelorarbeit kann nur einen Teil ausschöpfen von einem Optimierungsprozess, der noch viel Potential birgt.

Abstract

Let's say you are an engineer, who has specified a complex system. This system must be able to react to any possible enviromental event. Therefore, you itemized many scenarios, that are analized by a program. It detects sequences of events that your system can not handle in a satisfying way. How will you detect the part of your specification, that has to be corrected?

This bachelor thesis provides some awnsers to that. In fact, there already is an approach named „Play-Out“, which allows you to track every event step by step to find unwanted behavior of your system. However, they lack an oveview concerning a specific analysis of different states, in which undesired behavior occurs. Concepts improving this overview are presented. These Concepts inform you as an engineer about various relevant properties of the current and the reachable yet undesired states. They also offer a significant comparison of the different error conditions. Indeed, in the end it is up to the user to check whether a potential correction in the model is also realistic. But up to this point, much work is taken out of his hand. This bachelorthesis discusses only a small portion of an optimization process which still holds much potential.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **5**
2.1 Reaktive Systeme 5
2.2 Spezifikationen 7
2.3 Modal Sequence Diagrams 8
2.4 Counter-Play-Out 12
2.5 ScenarioTools 13

3 Konzepte **15**
3.1 Gegebene Daten 15
3.2 Erweiterung um Informationsanzeigen 16
3.3 Erweiterung um Fehlergruppierung 17
3.4 Weitere Betrachtungen 19

4 Prototyp **21**

5 Verwandte Arbeiten **23**
5.1 UPPAAL TIGA und PlayGo 23
5.2 Formale Fehler Modellierung 23

6 Résumé **25**
6.1 Zusammenfassung 25
6.2 Ausblick 26

A Alle Systemzustände der Beispielspezifikation **27**

B Counterstrategie der Beispielspezifikation **29**

C Counter-Play-Out der Beispielspezifikation **31**

Kapitel 1

Einleitung

Für komplexe, verteilte Systeme, wie zum Beispiel Produktionslinien, Flugzeuge oder moderne PKW, bedarf es einer Software, die diese reibungslos steuert und fehlerfreie Abläufe garantiert. Die Entwicklung einer solchen Software beginnt mit der informellen Spezifikation durch Ingenieure. Dieses geschieht meist in Form von Use Cases bzw. Szenarien, welche den Entwicklern eine gute Übersicht über einzelne Abläufe geben. Die Szenarien beschreiben Folgen von Ereignissen in der Umwelt des Systems und dessen erforderliche Reaktionen. Dadurch sind Szenarien intuitiv und können unabhängig voneinander betrachtet und erweitert werden [8]. Schwierig ist jedoch im Blick zu behalten, wie sich die einzelnen Szenarien beeinflussen oder ob sie sich sogar ausschließen. So sind beispielsweise zwei Anforderungen an ein autonomes Fahrzeug, dass es nicht abrupt bremst, wenn hinter ihm ein Auto dicht auffährt (1) und dass es vor einer roten Ampel hält (2). Doch wie soll sich das Fahrzeug verhalten, wenn es auf eine rote Ampel zu fährt und gleichzeitig jemand dicht auffährt? Dies ist eine mögliche Kombination von Umweltereignissen, die bei der separaten Formulierung der beiden Anforderungen nicht zwangsläufig offensichtlich ist. Nun ist eine vollständige Spezifikation, deren einzelne Szenarien sich nicht gegenseitig widersprechen, eine wichtige Grundlage für ein erfolgreiches Projekt: Das Beheben von Fehlern wird umso teurer, je fortgeschrittener der Entwicklungsprozess ist. Dies liegt daran, dass es zu kostspieligen Iterationen in der Entwicklung führen kann, wenn derartige Uneindeutigkeitenn bzw. Widersprüche in der Spezifikation erst spät erkannt werden.

Es wurden verschiedene Verfahren [4] [9] [17] entwickelt, um solche Widersprüche noch vor der Umsetzung der Spezifikation zu identifizieren. Abbildung 1.1 zeigt den Ablauf des Verfahrens, das in dieser Arbeit betrachtet wird. In der Abbildung sind Elemente, die für Dateien stehen, in einem hellblauen Rechteck und Prozesse in einem dunkelblauen Rechteck abgebildet. Die grauen Pfeile stellen den Ablauf des Verfahrens dar. Außerdem verfolgen die blauen Pfeile den kreativen Prozess des Entwicklers bezüglich etwaiger

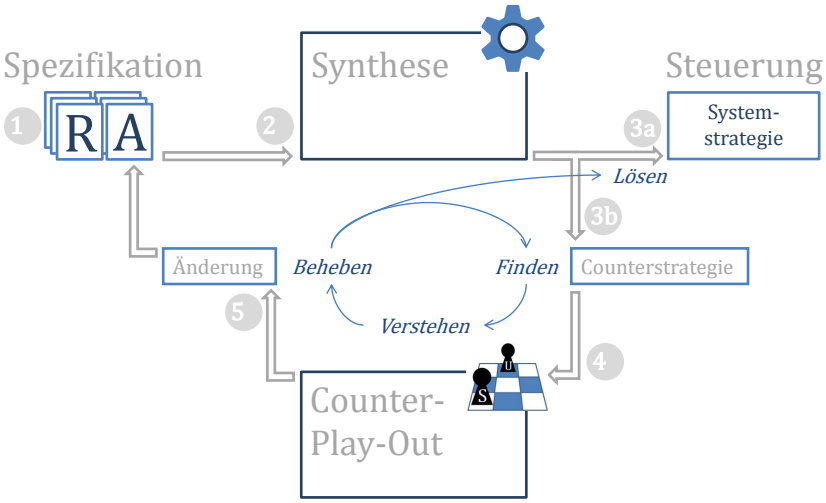


Abbildung 1.1: Von Systemspezifikation zu Systemsteuerung

Fehler beim Anwenden des Verfahrens. Dieses Verfahren kann in folgende fünf Schritte unterteilt werden.

Zunächst wird hierfür die szenariobasierte Spezifikation in Form von *Modal Sequence Diagrams* (*MSD*) [15] [5] erstellt (1). MSDs sind eine spezielle Art von Sequenzdiagrammen und beschreiben eindeutig für einzelne Sequenzen was geschehen kann, muss oder was nicht geschehen darf. Dabei teilen sich die MSDs in zwei Gruppen auf. Zum Einen in Anforderungen, die an das System gestellt werden und hier mit einem R wie *requirements* dargestellt sind. Und zum Anderen in Annahmen, wie sich die Umgebung verhält und auf die das System zu reagieren hat. Das A steht für die Annahmen bzw. *assumptions*.

Darauf folgt an Hand der Sequenzen die *Synthese* einer Steuerung (2). Als Prozess ist dieser Schritt als dunkelblauer Kasten dargestellt. Bei der Synthese wird die Spezifikation auf *Realisierbarkeit* geprüft. Eine Spezifikation ist realisierbar, wenn das System auf jede mögliche Sequenz von Umweltereignissen derart reagieren kann, sodass alle spezifizierten Anforderungen erfüllt werden (vgl. [13, Kap. 2.1]). Dieser Prozess kann auf zwei Arten ausgehen.

In Fall der Realisierbarkeit ist das Ergebnis der Synthese eine widerspruchsfreie *Systemsteuerung* (3a). Es bedarf keiner weiteren Arbeitsschritte. Anderenfalls wird eine *Counterstrategie* ermittelt (3b). Sie beruht auf einem spieltheoretischen Ansatz [7] [2] und wird hier kurz erläutert. Bei der Synthese wurde ein Zustandsgraph erstellt, der alle möglichen Kombinationen von Ereignissen und deren Folgen enthält. Bei den Zustandsübergängen, wird unterschieden, ob diese durch ein System- oder Umweltereignis ausgelöst wurden. Auf dieser Grundlage können die nötigen „Züge“ für die Counterstrategie berechnet werden, mit der die Umgebung das System sicher „besiegen“ kann.

Das System ist einerseits besiegt, wenn es keine Handlung mehr ausführen kann ohne eine Anforderung zu verletzen. Andererseits kann bereits vorher ein Umweltereignis gemäß den Annahmen eine Anforderung verletzen.

Es folgt eine Simulation, das *Counter-Play-Out* [19][14] (4). Hier kann der Entwickler erfahren, wie es zu Verletzungen kommen kann. In der Simulation tritt der Entwickler an die Stelle der Steuerung des Systems und der Computer ahmt die Umwelt des Systems nach. Mit Hilfe von Umwelteinflüssen gemäß der Counterstrategie wird die Steuerung bzw. der Entwickler zwangsläufig in einen *Fehlerzustand* geführt. In diesem Zustand ist also eine Verletzung vorgefallen.

Nach der Play-Out-gestützten Fehleranalyse kann der Entwickler Änderungen für die Spezifikation erarbeiten (5) und der Entwickler beginnt mit der angepassten Spezifikation erneut bei Schritt 1. Dieses Verfahren unterstützt also den Entwickler bei einem iterativen Prozess, der in Abb. 1.1 mit den blauen Pfeilen dargestellt ist. Ist eine Spezifikation gegeben, sollen etwaige Widersprüche gefunden, verstanden und behoben werden. Letztendlich werden so alle Widersprüche gelöst und man erhält die Systemsteuerung einer realisierbaren Spezifikation.

Je komplexer die Spezifikation ist, desto größer und unübersichtlicher wird der extrahierte Zustandsgraph. Dazu kann dieser auch mehr Fehlerzustände enthalten. Dies führt dazu, dass die Simulation sehr langwierig ist und möglicherweise viele Züge benötigt werden, bis ein Fehlerzustand erreicht ist. Der Entwickler verliert schnell die Übersicht über einzelne Fehlerzustände. Dieses Problem lässt sich damit lösen, dass intuitive Hilfen bereitgestellt werden, die dem Entwickler die Fehlerzustände aufbereiten. Trotz des Counter-Play-Outs ist es oft sehr knifflig die Szenarien an der richtigen Stelle zu korrigieren oder zu erweitern.

In dieser Bachelorarbeit werden Konzepte vorgestellt, die die Navigation zwischen den einzelnen Fehlerzuständen während der Simulation verbessern. So kommt man einer optimierten Unterstützung zur schnellen Beseitigung von Widersprüchen in Spezifikationen einen Schritt näher. Um die Orientierung zu verbessern, können bspw. folgende Informationen angezeigt und Funktionen bereitgestellt werden:

Informationen	① Minimale sowie maximale Anzahl von Zügen, bis ein Fehlerzustand erreicht wird ② Gesamtanzahl möglicher Fehler ③ Anzahl von Fehlern, die noch erreicht werden können
Funktionen	① Gruppieren von Fehlern nach bestimmten Mustern ② Wenige Schritte vor einen Fehlerzustand springen

Tabelle 1.1: Mögliche Orientierungshilfen im Counter-Play-Out

Nach den Erläuterungen der benötigten Grundlagen in Kapitel 2, folgt die

konzeptuelle Ausarbeitung verschiedener Hilfen in Kapitel 3. Diese werden anschließend als prototypische Erweiterung der Simulationsperspektive von ScenarioTools in Kapitel 4 vorgestellt. Daran schließt mit Kapitel 5 eine Darstellung verwandter Arbeiten an. Kapitel 6 beinhaltet abschließend eine Evaluation der erarbeiteten Lösungen mit Empfehlungen für weiteres Vorgehen außerhalb dieser Bachelorarbeit.

Kapitel 2

Grundlagen

Die beschriebene Problematik tritt bei so genannten reaktiven Systemen auf, die noch einmal in Abschnitt 2.1 erläutert und an einem Beispiel verdeutlicht werden. Diese Systeme werden zu Beginn ihres Entwicklungsprozesses umfassend spezifiziert. Näheres dazu in Abschnitt 2.2. Für die formale Spezifikation eines solchen Systems werden MSDs verwendet, welche in 2.3 beschrieben werden. Für ein derart spezifiziertes System kann, wie in der Einleitung erwähnt, eine Systemsteuerung oder im Fehlerfall eine Counterstrategie synthetisiert werden. Das Counter-Play-Out, in Abschnitt 2.4 erklärt, hilft beim Ergründen etwaiger Fehler. Anwenden lässt sich dies mit dem Eclipse-Plugin ScenarioTools, dessen für diese Arbeit relevanten Grundfunktionen in Abschnitt 2.5 vorgestellt werden.

2.1 Reaktive Systeme

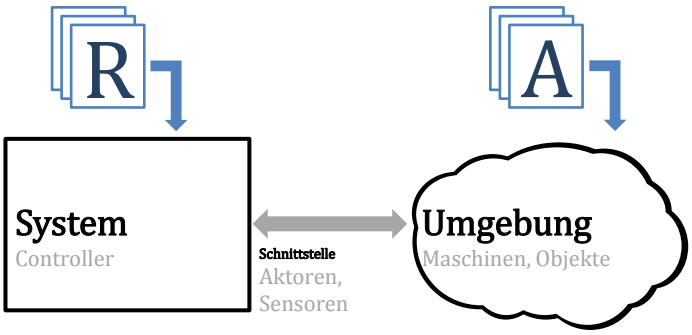


Abbildung 2.1: Beschreibung von System und dessen Umgebung

Ein reaktives System empfängt immer wieder Signale seiner Umgebung und beeinflusst daraufhin die Umgebung abhängig vom jeweiligen Signal und seinem Systemzustand. Wie in Abbildung 2.1 dargestellt, bekommt

das System diese Informationen über die Umwelt mittels Sensoren. Darauf reagiert das System, in dem es Aktoren ansteuert. Der Pfeil zwischen dem System und der Umgebung stellt diesen Datenaustausch dar. Was die angedeuteten Dokumente R und A bedeuten, wird in Abschnitt 2.2 erläutert. Die entscheidende Anforderung an reaktive Systeme ist die, dass sie zuverlässig und theoretisch endlos auf alle möglichen Einflüsse adäquat reagieren können müssen [18].

Verteilte Systeme sind eine Untergruppe von reaktiven Systemen. Oben beschriebene Systeme können auch kombiniert werden. So ist es heutzutage üblich, über 100 Chips in einem PKW separat zu verwenden. Die einen steuern Komponenten des Motors, die anderen das Audiosystem und wieder andere sind für Airbags, ABS, ESP usw. zuständig. Oder in einer Produktionslinie mit mehreren Prozessen hat jeder Roboter eine eigene Steuerung, welche zusammen wiederum von Toplevel-Steuerungen koordiniert werden. Je komplexer die zusammenhängenden Komponenten werden, desto schwieriger und wichtiger ist die fehlerfreie Entwicklung einer Steuerung hierfür. Fällt beispielsweise erst bei der Abnahme auf, dass sich zwei Roboterarme in die Quere kommen können, kostet die Korrektur viel Zeit und Geld. Daher ist eine korrekte Spezifikation unabdingbar.

Eine einfache Produktionszelle soll als fortlaufendes Beispiel dienen. Sie wurde in einem Artikel von Greenyer et al [12] dargestellt und ist hier in Abb. 2.2 skizziert. Zu sehen sind neben den beiden Fließbändern (feed

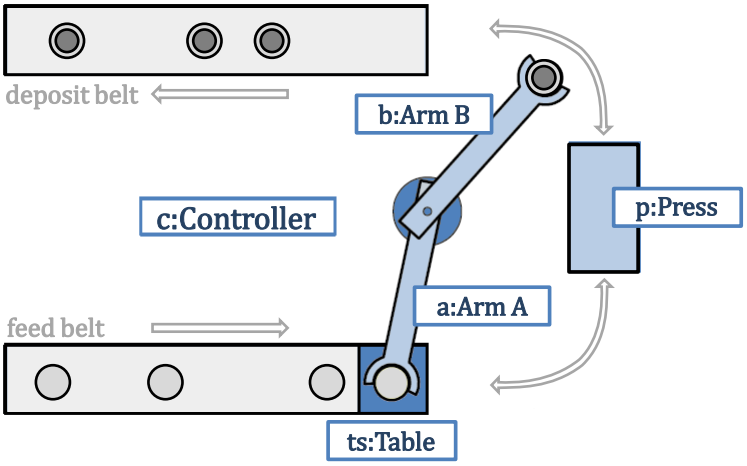


Abbildung 2.2: Skizze einer Produktionszelle, einem reaktiven System

belt und deposit belt), welche Rohlinge transportieren, auch die einzelnen Objekte des Systems und der Umgebung. In diesem Fall besteht das System

allein aus dem Controller. Ein Tisch mit einem Sensor, Arme A und B, sowie eine Presse gehören zur Umgebung. Die Aktoren der Steuerung können die Arme A und B eines Roboters bewegen, greifen und lösen lassen. Zudem kann der Presse das Signal zum Pressen gegeben werden. Die Sensoren wiederum melden, wenn ein neuer Rohling den Tisch erreicht, die Presse den Pressvorgang beendet oder wenn ein Arm seine Zielposition erreicht hat. Arm A bewegt sich zwischen Tisch und Presse, während Arm B zwischen Presse und Ablageband pendeln kann. Die Funktion dieser Anlage ist denkbar einfach: ankommende Rohlinge sollen gepresst und wieder abtransportiert werden. Wie dieses System spezifiziert werden kann, wird nun erklärt.

2.2 Spezifikationen

In dieser Arbeit werden szenariobasierte Spezifikationen betrachtet. Szenariobasiert bedeutet, es werden einzelne Abläufe beschrieben. In diesen Abläufen kommen mehrere Objekte vor, die daran beteiligt sind. Jedoch wird nicht die komplette Funktionsweise eines einzelnen Objektes beschrieben. Dies wäre eine alternative Herangehensweise, welche meist als nicht intuitiv beschrieben wird (vgl. [16, Kap. 1]). Grund dafür ist, dass man dabei deutlich mehr über ein System wissen muss als dessen geforderte Sequenzen. Eben diese Sequenzen stehen am Anfang einer Spezifikation. Sie sind der Zweck, den ein System erfüllen soll. Der Zweck der Beispielproduktionszelle aus Abschnitt 2.1 ist, dass Rohlinge gepresst werden. Die Objekte Sensoren, Roboterarme und Presse sind dabei nur Mittel zum Zweck; daher können sie erst definiert werden, wenn alle Szenarien bekannt sind. Um ein Objekt vollständig beschreiben zu können, benötigt man also Wissen darüber, in welchen Sequenzen es involviert ist und auf welche Ereignisse es wie reagieren muss.

Wie schon in der Einleitung erwähnt und in Abb. 2.1 dargestellt, spezifizieren die Anforderungen (requirements) wie das System mit seiner Umgebung interagieren soll.

Für eine Spezifizierung der Produktionszelle (Abb. 2.2) werden zunächst dessen Objekte und ihre Eigenschaften definiert. Anschließend werden die Anforderungen formuliert. In Tabelle 2.1 werden die Aufgaben von Arm A (R1), Arm B (R3) und der Presse (R2) beschrieben. Jeder dieser Abläufe für sich, kann nicht weiter unterteilt werden, da genau diese Sequenz abgearbeitet werden muss, bevor sie wieder von vorn begonnen werden kann. Umgekehrt muss formuliert werden, welche Ereignisse in der Umwelt auftreten können. Ein Umweltereignis ist in diesem Beispiel unter anderem, dass ein neuer Rohling angekommen ist. Das System muss nur damit umgehen können, dass ein weiterer Rohling ankommt, wenn Arm A zum Tisch zurückgekehrt ist. Somit benötigt man die Annahme (assumption), dass der nächste Rohling auch erst dann (oder später) ankommt. Darüber hinaus wartet das System,

R1	Wenn ein Rohling am Tisch angekommen ist, muss Arm A diesen aufnehmen, zur Presse fahren, ihn dort loslassen und zurück zum Tisch fahren.
R2	Wenn ein Rohling in die Presse gelegt wurde, muss diese pressen. Nach dem Pressvorgang muss Arm B den gepressten Rohling aufnehmen.
R3	Wenn Arm B einen Rohling aufgenommen hat, muss er zum Ablageband fahren, den Rohling dort loslassen und zurück zur Presse fahren.

Tabelle 2.1: Anforderungen an die Produktionszelle aus Abb. 2.2

nachdem es einen Arm in eine bestimmte Richtung geschickt hat, auf die Rückmeldung eines Sensors, dass der Arm dort angekommen ist. Dass jeder Arm auch irgendwann an seinem Ziel ankommt muss ebenfalls in den Annahmen spezifiziert sein. Genauso verhält es sich beim Pressvorgang. Gemeinsam bilden Anforderungen und Annahmen die Spezifikation für ein System. Auch in Abbildung 2.1 sind diese als mit R und A betitelte Dokumente dargestellt.

Wird die Spezifikation in einer maschinenlesbaren Semantik verfasst, kann diese durch entsprechende Algorithmen wie in der Einleitung beschrieben geprüft werden. Im Folgenden wird erklärt, wie eine solche Semantik definiert ist und wie diese eine Prüfung auf Realisierbarkeit zulässt.

2.3 Modal Sequence Diagrams

MSDs wurden 2008 von Harel und Maoz entwickelt [15]. Diese Diagramme basieren auf UML und bestehen wie übliche Sequenzdiagramme aus Objekten, die durch jeweils eine *Lebenslinie* (lifeline) dargestellt werden. Die Interaktion dieser Objekte erfolgt über *Nachrichten* (messages) entlang der Lebenslinie. Die Gesamtheit der Objekte, die sich einander Nachrichten senden können, wird *Objektsystem* genannt. Die Objekte in einem Objektsystem werden in Umweltobjekt und Systemobjekt unterschieden (vgl. Abschnitt 2.1).

Das Versenden einer Nachricht im Objektsystem zwischen zwei Objekten wird auch *Ereignis* genannt. Ist hier der Absender ein Systemobjekt, spricht man von einer *Systemnachricht*. Analog dazu werden *Umweltnachrichten* von Umweltobjekten verschickt. Ein Ereignis umfasst sowohl das Senden als auch das Empfangen einer Nachricht.

Das Innovative an diesen MSDs ist deren Aussagekraft, die mit der Semantik der versendeten Nachrichten erreicht wird. Dadurch sagen MSDs eindeutig aus, was geschehen muss, darf oder nicht darf. Nachrichten haben eine *Temperatur* (temperature: hot or cold) und eine *Ausführungsart* (execution kind: executed or monitored). Diese Eigenschaften können wie in der

Tabelle 2.2 abgebildet kombiniert werden.

message semantics	cold (can be „violated“)	hot (must not be „violated“)
monitored (may happen)	c,m	h,m
executed (must happen)	c,e	h,e

Tabelle 2.2: Semantik der Nachrichten in MSDs, vgl. [13]

Der blaue, gestrichelte steht für eine *kalte*, *überwachte* Nachricht, die verletzt werden kann und von der auch nicht gefordert ist, dass diese irgendwann eintritt. Ein roter, gestrichelter Pfeil steht für eine *heiße*, überwachte Nachricht, die im Unterschied dazu nicht verletzt werden darf, von der aber auch nicht gefordert wird (oder werden kann), dass sie irgendwann eintritt. Kalte, *ausgeführte* Nachrichten werden durch einen blauen, durchgehenden Pfeil dargestellt. Sie müssen zum einen geschehen, dürfen aber verletzt werden. Heiße, ausgeführte Nachrichten (roter, durchgehender Pfeil) müssen irgendwann versendet werden und dürfen nicht verletzt werden (vgl. [12]. Wie Verletzungen zustande kommen und was deren Konsequenzen sind, wird weiter unten erklärt.

Anforderungen und Annahmen stellen eine Art Spielregeln für das System sowie die Umgebung dar. Die Spielregeln für das System können nur vorgeben, wie selbiges zu reagieren hat. Sie können aber nicht garantieren, dass ein Umweltereignis eintritt. Umgekehrt gilt das gleiche. Daher sind grundsätzlich in Anforderungen alle Umweltnachrichten überwacht. In Annahmen sind umgekehrt alle Systemnachrichten überwacht. Außerdem ist in jedem MSD die erste Nachricht kalt und überwacht.

Innerhalb des Objektsystems wird eine Nachricht eindeutig durch dessen Namen, den Empfänger und den Absender identifiziert. Innerhalb eines MSDs kommen noch die beiden Attribute Temperatur und Ausführungsart hinzu.

Eine Sequenz von versendeten Nachrichten wird *Durchlauf* (run) genannt. Während eines Durchlaufs werden Ereignisse wie folgt mit den MSDs einer Spezifikation abgeglichen. Eine *aktive Kopie* des entsprechenden MSDs wird erzeugt, indem dessen erste Nachricht versendet wird. Dies kann geschehen, wenn sie in einer anderen aktiven Kopie gefordert und ausgeführt wird, oder von der Umwelt ausgelöst wurde.

Cuts zeigen an, wie weit MSDs bereits abgearbeitet wurden, in dem sie die Mengen ebendieser bereits abgearbeiteten Nachrichten sind. Hierbei ist noch folgendes zu beachten. Nach dem Aktivieren einer MSD-Kopie, wird die darauf folgende Nachricht im MSD *aktiviert*. Der Cut beinhaltet nun

die erste Nachricht des MSDs. Stimmt die Nachricht eines Ereignisses in Namen, Empfänger und Absender mit der aktivierten Nachricht überein, werden diese *unifiziert*. Dadurch wird die betroffene Nachricht dem Cut hinzugefügt und ist nicht mehr freigeschaltet. Der Cut springt also zur nächsten, nun freigeschalteten Nachricht. Ein Cut hat die gleiche Temperatur und Ausführungsart wie die darauf folgende Nachricht und wird aktiviert, wenn die vorherige Nachricht versendet wurde. Eine aktive Kopie terminiert, wenn alle Nachrichten im Cut enthalten sind, bzw. wenn die letzte Nachricht des entsprechenden MSD mit einer System- oder Umweltnachricht unifiziert wurde.

Ist eine aktivierte Nachricht heiß, muss eine solche versendet werden, bevor irgendeine anderes Ereignis eintritt, das mit einer Nachricht aus der selben Kopie unifiziert werden kann. Andernfalls tritt eine *Sicherheitsverletzung* auf. Analog dazu kann bei einer kalten, aktivierten Nachricht eine *kalte Verletzung* auftreten. Ein durchgezogener Pfeil besagt, dass ein Ereignis mit dieser Nachricht irgendwann eintreten muss. Ist so eine Nachricht aktiviert, kann es zu einer *Lebendigkeitsverletzung* kommen, wenn der dazugehörige Cut im weiteren Verlauf des Durchlaufs nicht fortschreitet (vgl. [12]). Es wird zwischen einer Verletzung in einer Anforderung oder einer Annahme unterschieden. Eine Sicherheitsverletzung wird im Folgenden mit SV und eine Lebendigkeitsverletzung LV genannt. Verletzungen sind ausschlaggebend dafür, ob eine Spezifikation als realisierbar eingestuft wird oder nicht. Mehr dazu wird im Folgeabschnitt 2.4 erklärt.

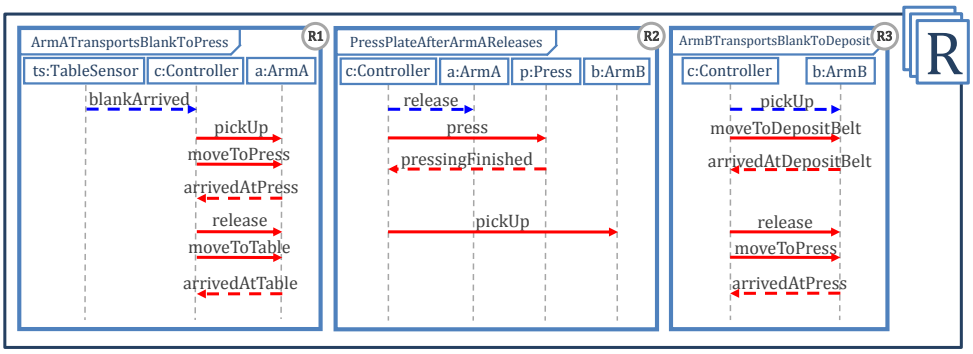


Abbildung 2.3: Anforderungen R1-R3 aus Tab. 2.1

An Hand dieser Informationen können nun für die Anforderungen der Produktionszelle aus 2.1 MSDs erstellt werden. Die im Folgenden betrachtete Umsetzung ist in Abb. 2.3 zu sehen. R1 setzt die Anforderung um, dass ein Rohling aufgenommen, zur Presse gefahren und losgelassen wird der Arm anschließend zurück zum Tisch fährt. Hierbei wird jede einzelne Aktion, die erfolgen muss als roter, durchgezogener (hot, executed) Pfeil dargestellt. Ausgelöst wird dieser Ablauf, wie in Tabelle 2.1 beschrieben,

durch die Nachricht des Tischsensors. Da dies die erste Nachricht des MSDs ist, ist diese kalt und überwacht. Um sicher zu gehen, dass der Rohling erst losgelassen wird, wenn Arm A an der Presse angekommen ist, muss zuvor noch die entsprechende Bestätigung an das System gesendet werden (*arrivedAtPress*). Diese ist allerdings eine Umweltnachricht; daher kann diese in den Regeln für das System nicht als sicher eintretende Nachricht angenommen werden. Der Pfeil ist also rot (darf nicht verletzt werden) und gestrichelt (passiert vielleicht). Mit der gleichen Begründung ist die Nachricht *arrivedAtTable* ebenfalls heiß und überwacht.

Auf die gleiche Weise werden die Anforderungen R2 und R3 interpretiert. Aus einem solchen Satz sollte ein MSD quasi ablesbar sein. Bei R2 geht es bspw. wie in Tabelle 2.3. Die Abbildung 2.4 enthält Annahmen über

Wenn ein Rohling in die Presse gelegt wurde	cold, monitored: release [c->a]
, muss diese pressen.	hot, executed: press [c->p]
Nach dem Pressvorgang	hot, monitored: pressingFinished [p->c]
muss Arm B den gepressten Rohling aufnehmen.	hot, executed: pickUp [c->b]

Tabelle 2.3: Anforderung 2 wird in MSD Semantik übertragen

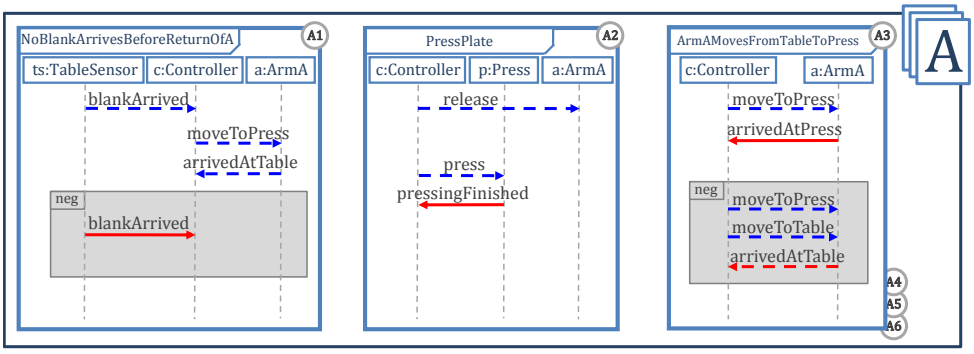


Abbildung 2.4: Annahmen A1-A3 von insg. 6, wobei A4-A6 der 3. sehr ähnlich sind

die Umwelt, wie sie im Abschnitt 2.2 beschrieben wurden. A1 und A3-A6 enthalten Kästen, die mit *neg* betitelt sind. Solch ein Kasten ist eine Negation und die darin enthaltenen Nachrichten werden für die gesamte Sequenz verboten, in der eine Kopie der MSD aktiv ist, die diese Negation enthält. „NoBlankArrivesBeforeReturnOfA“ beschreibt also die Annahme, dass zwischen den Umweltereignissen *blankArrived* und *arrivedAtTable* kein

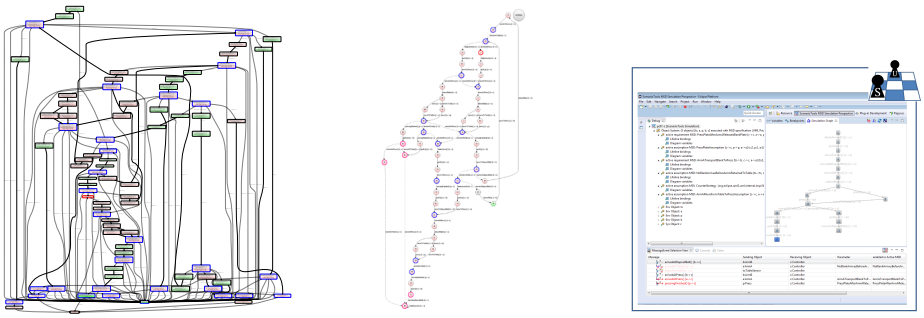
weiteres *blankArrived* versendet wird. Die Annahme „PressPlate“ besagt, dass nach der Systemnachricht *release*, nach der auch *press* folgte, muss irgendwann die Umweltnachricht *pressingFinished* erfolgen. So verhält es sich auch mit den Annahmen über die Bewegungen der Roboterarme. Wurde ein Arm zu einem neuen Ziel geschickt, muss es auch irgendwann dort ankommen. In der Abbildung 2.4 wurde dies exemplarisch für den Vorgang Arm A bewegt sich vom Tisch zu Presse dargestellt. Hier sieht man auch eine Negation, die verbieten, dass während Arm A auf dem Weg vom Tisch zur Presse ist, keine Systemnachricht versendet werden darf, die Arm A erneut ein Ziel (Tisch oder Presse) vorgibt. Sollte dies trotzdem vorkommen, hätte dieses Ereignis eine kalte Verletzung zur Folge. Das Umweltereignis *arrivedAtTable*, welches in dieser Situation aussagte, dass Arm A auf dem Weg zur Presse am Tisch angekommen sei, führte dagegen zu einer Sicherheitsverletzung. Die anderen drei Annahmen beinhalten berücksichtigen analog, nur mit jeweils dem anderen geforderten und verbotenen Ziel.

2.4 Counter-Play-Out

Das Counter-Play-Out beruht auf einer Spielstrategie für Büchi-Spiele, einer eingeschränkten Form von Zwei-Spieler-Spielen. Mit einer solchen Strategie kann einer von beiden Spielern sicher gewinnen. Hierfür muss zunächst bestimmt werden, welcher Spieler mit Sicherheit gewinnen kann. Das System gewinnt, wenn dessen Steuerung realisierbar ist. Ist dem nicht so, gewinnt die Umgebung und das System verliert. Die Bedingung für die Realisierbarkeit eines reaktiven Systems lautet: „Ein System ist realisierbar, wenn ein unendlich langer Durchlauf möglich ist, in welchem immer wieder ein Ziel-Zustand erreicht wird.“

Ziel-Zustände sind wie folgt Definiert (vgl. [12, Kap. 3.2]): Entweder ist in keinem Anforderungs-MSD eine SV aufgetreten und in keiner aktiven Kopie einer Anforderungs-MSD ist der Cut ausgeführt (1) oder es ist eine SV in einem Annahmen-MSD aufgetreten (2) oder mindestens eine Annahmen-MSD befindet sich noch in einem ausgeführten Cut (3). Existieren noch Widersprüche in der Spezifikation, gibt es also umgekehrt Sequenzen von Umweltereignissen, durch die eine Sicherheitsverletzung in den Anforderungen, nicht aber in den Annahmen auftritt. Ob die Realisierbarkeitsbedingung erfüllt wird, prüft während der Synthese der On-the-fly Algorithmus für Büchi-Spiele von Cassez et al. [6]. Dieser Algorithmus wird in einem Paper von Greenyer et al. [12] explizit für die Synthese von Systemen vorgestellt.

Mit einer derart ermittelten *Counterstrategie* kann nun im Counter-Play-Out der Anwender Schritt für Schritt nachverfolgen, durch welche Umweltereignisse das System besiegt wird.



(a) Systemzustände (b) Counterstrategie (c) Play-Out

Abbildung 2.5: Schritte zur Fehleridentifizierung vgl. Anhänge A-C

2.5 ScenarioTools

Dieses Eclipse Plug-In steht unter `http://scenariotools.org/download/` zum Download zur Verfügung. Es realisiert das Counter-Play-Out, indem man interaktiv einen Zustandsgraphen (Historygraph) entwickelt [14]. Nach dem Erzeugen von MSDs und der Beschreibung des Objektsystems, wird zunächst ein Zustandsgraph für alle möglichen Durchläufe exploriert. Das Beispielsystem soll lediglich drei Anforderungen erfüllen und man erhält für dessen Spezifikation bereits einen sehr umfangreichen Zustandsgraphen, der in Abb. 2.5a zu sehen ist.

Eine Counterstrategie, wie in Abb. 2.5b besteht aus einem Teilgraphen davon. Dieser Teilgraph muss allerdings nicht alle möglichen Pfade enthalten, durch die das System verliert. Genauer ist die Counterstrategie in Abb. B.1 zu sehen. Es gibt einen rot-umrahmten Knoten, welcher der Anfangszustand ist. Rosa umrandete Knoten bedeuten, dass hier das System verloren hat. Ist ein Knoten grün gefüllt, steht diese für einen Zustand, von dem aus das System in einen Zustand kommt, in dem kein Cut ausgeführt ist. Grüne Knoten mit einem blauen Rand stehen für einen solchen Zustand. Rote Knoten dagegen bedeuten, dass das System von hier aus verlieren kann. Kann in einem Zustand die Umwelt beeinflussen, ob das System verliert, ist dieser Knoten rot gefüllt und blau umrahmt (vgl. [14, Kap. 2.2]).

MessageEvent Selection View | Console | Tasks

Message	Sending Object	Receiving Object	Parameter	enabled in Active MSD
arrivedAtDepositBelt() [b->c]	b:ArmB	c:Controller		
arrivedAtTable() [a->c]	a:ArmA	c:Controller	NoBlankArrivesBeforeAr...	NoBlankArrivesBeforeAr...
blankArrived() [ts->c]	ts:TableSensor	c:Controller		
arrivedAtPress() [b->c]	b:ArmB	c:Controller		
arrivedAtPress() [a->c]	a:ArmA	c:Controller	ArmATransportBlankToP...	ArmATransportBlankToP...
pressingFinished() [p->c]	p:Press	c:Controller	PressPlateAfterArmARele...	PressPlateAfterArmARele...

Abbildung 2.6: MessageEvent Selection View

Die Ausführung des Counter-Play-Outs mit ScenarioTools ist in Abb. 2.5c zu sehen. Die im aktuellen Zustand versendbaren Nachrichten werden in der $MessageEventSelectionView$ angezeigt. Diese ist in Abb. 2.6 noch einmal genauer zu sehen. Unter anderem durch die verschiedenen Icons vor den Nachrichten erfährt der Anwender näheres zur jeweiligen Nachricht. Ob eine kalte Verletzung oder Sicherheitsverletzung ausgelöst wird, wird durch einen blauen Blitz oder ein rotes Verbotszeichen dargestellt. Unterteilt wird dies in einer Spalte für Annahmen und eine für Anforderungen.

An Hand der umfangreichen Graphen aus Abbildung 2.5 lässt sich schnell erkennen, dass ein Counter-Play-Out mit zunehmender Komplexität des Spezifizierten System sehr mühselig wird, da dann auch die Anzahl der Zustände stark zunimmt.

Kapitel 3

Konzepte

Um also mehr Übersichtlichkeit zu schaffen, wird in Abschnitt 3.1 ermittelt, welche Informationen über das System und die Counterstrategie zur Verfügung stehen. Anschließend stelle ich konkrete Hilfestellungen vor, die diese Informationen aufbereiten.

3.1 Gegebene Daten

Für die Analyse stehen die drei Elemente aus Abb. 2.5 zur Verfügung: der Zustandsgraph, der daraus errechnete Strategiegraph und das Counter-Play-Out, in dem Schritt für Schritt der Strategiegraph durchlaufen werden kann. Im Zustandsgraphen sind für jeden Zustand Informationen über dessen aktive MSDs gespeichert. Darüber hinaus enthält jeder Zustand die Information, ob von ihm aus das System gewinnen oder verlieren kann und ob der Zustand ein Ziel-Zustand ist oder eine oder mehrere SV aufgetreten ist.

Hieraus lässt sich die Menge S aller Zustände, in denen eine SV aufgetreten ist, bestimmen. Außerdem kann die Menge L aller Zustände in einer LV berechnet werden. Zustände, die in einer LV sind, befinden sich innerhalb einer *Strongly-Connected-Component*, in der kein Ziel-Zustand enthalten ist.

Solche Strongly-Connected-Components sind Teilgraphen, in welchen von jedem beliebigen Knoten jeder andere Knoten erreicht werden kann. Diese können zum Beispiel mit dem Algorithmus von Tarjan [20, S.157] berechnet werden. Um den Algorithmus anwenden zu können, muss sichergestellt werden, dass dabei Transitionen zu Ziel-Zuständen nicht berücksichtigt werden.

Die folgenden Vorschläge zur Erweiterung 3.2 und 3.3 beschreiben Hilfen für das Counter-Play-Out. Die Informationsanzeigen bereiten vorwiegend die Informationen aus den Zustandsmengen S und L auf. Die darauf folgenden Gruppierungsvorschläge berücksichtigen zusätzlich die jeweiligen Cuts.

3.2 Erweiterung um Informationsanzeigen

Nun stellt man sich die Frage, welche Möglichkeiten es gibt, in einem Counter-Play-Out von einem derzeitigen Zustand zu verlieren. Zunächst werden diese einmal dargestellt.

Es kann Zustände geben, von denen aus man nicht in eine SV aber auch nicht zu einem Ziel-Zustand gelangen kann. Ist dies der Fall, befindet sich dieser Zustand vor oder in einer LV. Man kann sich auch in oder kurz vor einer LV befinden und trotzdem mindestens eine SV erreichen. Die letzte Möglichkeit ist, es sind keine LV und nur SVen erreichbar. Gerade wurden drei Eigenschaften eines Zustandes kombiniert, um Aussagen über mögliche *Verletzungsverläufe* zu treffen. Diese Eigenschaften sind *SV erreichbar* (Safety Violation Will Occur), *LV erreichbar* (Liveness Violation Will Occur) und *bereits in LV* (Already in Liveness Violation).

Die drei Eigenschaften lassen sich wie folgt definieren. Es sei $x \in V$ und $\forall z \in S, L : z \in V$ für einen einem Zustandsgraphen $G = (V, E)$.

Definition 1 (Safety Violation Will Occur)**.** Von x ist eine SV erreichbar, genau dann wenn ein Pfad von x zu y existiert, mit $y \in L$.

Definition 2 (Already in Liveness Violation)**.** x ist bereits in einer LV, genau dann wenn $x \in L$.

Definition 3 (Liveness Violation Will Occur)**.** Von x ist eine LV erreichbar, genau dann wenn ein Pfad von x zu y existiert mit $y \in L$, $x \notin L$.

Werden diese drei Eigenschaften vor einem Play-Out für jeden Zustand berechnet, können diese während des Play-Outs bspw. durch drei Lampen angezeigt werden.

SVW	LVW	LV	Kombination
●	○	○	Savety Violation Will Occur
○	●	○	Liveness Violation Will Occur
○	○	●	Already in Liveness Violation
●	●	○	Savety Violation or Liveness Violation Will Occur
●	○	●	Savety Violation Will Occur and Already in LV

Tabelle 3.1: Kombinationsmöglichkeiten

Die möglichen Kombinationen der Lampen SVW (Safety Violation Will Occur), LVW (Liveness Violation Will Occur) und LV (Already in Liveness Violation) sind noch einmal in Tabelle 3.1 aufgelistet.

Zusätzlich kann für jeden Zustand bestimmt werden, wie viele Schritte noch minimal und maximal nötig sind, um in eine SV zu gelangen. Dies kann durch eine Breitensuche realisiert werden. Man beachte, dass in oder vor einer LV die maximale Anzahl möglicher Schritte bis zu einer SV unendlich ist.

Mit diesen ist dem Anwender schon viel geholfen. Nun kann er während des Play-Outs erkennen, ob die kommenden, noch möglichen Systemzustände für ihn interessant sein könnten, oder ob er einen anderen Weg ausprobieren soll. Der kürzeste Weg zu einer SV kann nun auch leichter gefunden werden. Außerdem sind LVen erkennbar, bevor man sie komplett durchgegangen ist. Dies ist hilfreich, wenn man eigentlich zunächst die Fehlerzustände untersuchen möchte. Über eben diese ist allerdings noch nicht allzu viel bekannt.

Hilfreich wäre es also, wenn im Counter-Play-Out noch eine Funktion bereit gestellt wird, die Informationen über alle Fehlerzustände bündelt. Dadurch kann der Anwender Fehler gezielter untersuchen.

3.3 Erweiterung um Fehlergruppierung

Zustände unterscheiden sich durch die Cuts in den jeweilig aktiven MSDs. So können in zwei Zuständen die gleichen MSDs aktiv sein, jedoch enthält in dem einen Zustand mindestens ein freigeschalteter Cut mehr Nachrichten (vgl. 2.4). Durch diese Informationen lassen sich Aussagen über Ähnlichkeiten von Fehlern treffen. Hierfür werden zwei aufeinander folgende Zustände betrachtet, die zunächst definiert werden.

Ein *Zustand unmittelbar vor einer Verletzung* (State Of Imminent Violation) ist ein Zustand, in dem noch keine Verletzung aufgetreten ist (1) und im nächsten Schritt eine Sicherheitsverletzung der Anforderungen (2) auftritt und das System dann nicht mehr gewinnen kann (3). Der dann erreichte Zustand heißt *Verletzungszustand*.

Darüber hinaus bedarf es für eine Verletzung in einem Zustand mindestens ein aktives MSD, in welchem die Verletzung auftritt. Als *Verletztes MSD* wird ein MSD bezeichnet, welches im Zustand unmittelbar vor einer Verletzung aktiv ist und im Verletzungszustand durch eine SV beendet wurde.

Eine *Verletzende Nachricht* ist die Nachricht, die das System von einem Zustand unmittelbar vor einer Verletzung in den Verletzungszustand führt. In einem Zustand unmittelbar vor einer Verletzung ist ein darin aktives MSD ein *Verletzendes MSD*, wenn der Cut im MSD vor der verletzenden Nachricht steht.

Definition 4 (Gruppierung nach MSD-Tupel)**.** Eine MSD-Gruppe wird aus den Zuständen unmittelbar vor einer Verletzung gebildet, bei denen die Tupel aus Verletzendem MSD und Verletztem MSD identisch sind.

Definition 5 (Verfeinerte Gruppierung nach Cut-Tupel)**.** Eine Cut-Gruppe wird aus Zuständen einer MSD-Gruppe gebildet, bei denen die Cuts in Verletzendem und Verletztem MSD identisch sind.

Diese Gruppierungsmöglichkeiten werden nun an Hand der Beispielspezifikation vorgestellt. Tabelle 3.2 zeigt hierfür zeilenweise alle Zustände

unmittelbar vor einer Verletzung auf. Die Counterstrategie auf die sich die Zustandsnummern 54, 60 und 72 in der Tabelle beziehen, ist in B abgebildet. Die Spalten stehen für die einzelnen Anforderungs-MSDs R1-R3 aus Abb. 2.3 und rechts für die Annahmen-MSDs A1-A6 aus Abb. 2.4. Wobei A4-A6 in der Abbildung nur angedeutet und hier nicht relevant sind. A4 ist die Annahme „ArmAMovesFromPressToTable“, A5 ist „ArmBMovesFromPressToDepositBelt“ und A6 ist „ArmBMovesFromDepositBeltToPress“. Die Tabelle gibt für jeden aufgeführten Zustand an, in welchem Cut sich dessen aktive MSDs befinden. Im Zustand mit der Nummer 54 ist der Cut in Anforderung R2 an der Stelle $c6$. Das bedeutet, dass der Cut nach der sechsten Nachricht, *moveToTable*, und somit vor *arrivedAtTable* steht. Alle Zustände in der Tabelle befinden sich in Anforderung R2 im Cut $c2$, welcher zwischen *pressingFinished* und *pickUp* ist. In den Zuständen 60 und 72 wartet das System auf Grund des Cuts in R3 auf das Umgebungsereignis *arrivedAtPress*, welches in der Tabelle als Cut $c5$ symbolisiert ist. Im Zustand 54 befindet sich das MSD R2 in Cut $c2$ und wartet somit auf *arrivedAtDepositBelt*.

In dem Graphen der Counterstrategie sieht man, dass die Verletzende Nachricht *pickUp* ist, da diese jeweils von den Zuständen 54, 60, 72 zu deren Verletzungszustand führen. Daraus ergibt sich für alle drei Zustände R2 als Verletzendes MSD. Weiter gilt für alle drei Zustände, dass sich in ihnen R3 in einem heißen Cut befindet und durch *pickUp* mit einer SV beendet wird. Somit stimmen bei allen drei Zuständen unmittelbar vor einer Verletzung das Verletzende und Verletzte MSD überein. Sie bilden also eine MSD-Gruppe. Zustand 60 und Zustand 72 bilden sogar eine Cut-Gruppe.

states\ MSDs	R1	R2	R3	A1	A2	A3	A4	A5	A6
54 (56)	-	c3	c2	-	-	-	-	c1	-
60 (51)	c6	c3	c5	c2	-	-	c1	-	c1
72 (62)	-	c3	c5	-	-	-	-	-	c1

Tabelle 3.2: Vergleich der Zustände unmittelbar vor einer Verletzung

54 (56): Nr. in B.1 (Nr. in A.1)
c3 : Cut nach dritter Nachricht
R1-R3 aus Abb. 2.3 und A1-A6 u. a. aus Abb. 2.4

In der Beispielspezifikation sind also alle drei Zustände unmittelbar vor einer Verletzung in der gleichen MSD-Gruppe. Somit liegt die Vermutung nahe, dass alle durch den gleichen Widerspruch ausgelöst werden und lediglich in einem anderen Kontext stehen. Der Entwickler erhält beim Counter-Play-Out Erkenntnisse über eine der drei Verletzungen, in dem er bis zu dem

entsprechenden Zustand simuliert hat. Nun kann er mit dem Wissen über dessen MSD-Gruppenzugehörigkeit auch Rückschlüsse auf weitere Zustände dieser Gruppe ziehen und muss diese ggf. nicht mehr simulieren.

3.4 Weitere Betrachtungen

Nun könnte der Entwickler entscheiden, welche Fehlerzustände er sich genauer ansehen möchte. Sehr komfortabel wäre eine Funktion, die daraufhin anbietet, das Counter-Play-Out automatisch bis kurz vor diesen Zustand zu spielen. Dadurch ergibt sich der Vorteil, dass man auch bei umfangreichen Graphen schnell zu der entscheidenden Situation kommt, an der der Fehler auftritt.

Zusätzlich könnte auch berücksichtigt werden, ob die verletzende Nachricht System- oder Umgebungsnachricht ist. Wie dies aussehen könnte, habe ich in dieser Arbeit nicht mehr ausarbeiten können. Außerdem ist es möglich, dass mehrere MSDs eines Zustandes Verletzende (oder Verletzte) MSDs sind. Auch hier kann die Gruppierung also noch weiter verfeinert werden.

Kapitel 4

Prototyp

In diesem Kapitel wird nun kurz erläutert, wie die gerade vorgestellten Konzepte in die Simulationsumgebung von ScenarioTools integriert werden können. Das MockUp in Abbildung 4.1 skizziert die *ScenarioTools MSD Simulation Perspective*, die auch in Abb. C.1 zu sehen ist. Beide stellen das Counter-Play-Out mit der gleichen Counterstrategie für die Beispielspezifikation dar. Das MockUp enthält folgende zwei Erweiterungen.

Über dem *Historygraph* ist ein zusätzlicher Balken zu sehen. Darin sind drei Kreise, wovon einer ausgefüllt ist. Unter dem ausgefüllten Kreis steht *SVW*, was bereits in der Tabelle 3.1 dafür stand, dass eine SV erreichbar ist. Von dem Zustand, in dem sich die Simulation gerade befindet, ist es also möglich, in eine SV zu gelangen. Die Kreise über *LVW* und *LV* sind nicht ausgefüllt. Dies bedeutet, dass in diesem Zustand und auch in den folgenden LVen keine Rolle spielen. In einem Textfeld neben dieser Anzeige, gibt das Tool aus, wie viele Züge noch minimal und maximal vorgenommen werden können, bis eine SV erreicht wird. Wäre LV erreichbar, stünden ebenso die Anzahlen hierfür in dem Textfeld. In diesem Fall ist es sogar möglich, ohne in eine LV zu gelangen, nicht in eine SV zu gelangen.

Auf der linken Bildhälfte zwischen *Debug View* und *MessageEvent Selection View* wurde die *Sort-Function View* eingefügt. Sie bietet zwei Funktionen an. Zum einen kann der Graph der Counterstrategie nach Cut-Gruppen (*Cut-Groups*) sortiert werden. Hierfür dient der Button mit selbiger Aufschrift. Zu sehen ist, dass die Gruppierung nach MSD-Gruppen aktiv ist. Wie im vorangegangenen Kapitel erklärt, sind alle Zustände unmittelbar vor einer Verletzung, die mit der betrachteten Counterstrategie erreicht werden können, in einer MSD-Gruppe. Die Nummer der Zustände beziehen sich allerdings auf die Zustandsnummern des Systemzustandsgraphen (vgl. A.1). Das liegt daran, dass die Nummerierung der Zustände im Historygraph von Reihenfolge anhängen, wie sie im Play-Out ausgeführt werden.

Der Bezug von den gruppierte Zuständen und dem Historygraph muss daher noch deutlicher hergestellt werden. Zum Beispiel kann dies geschehen,

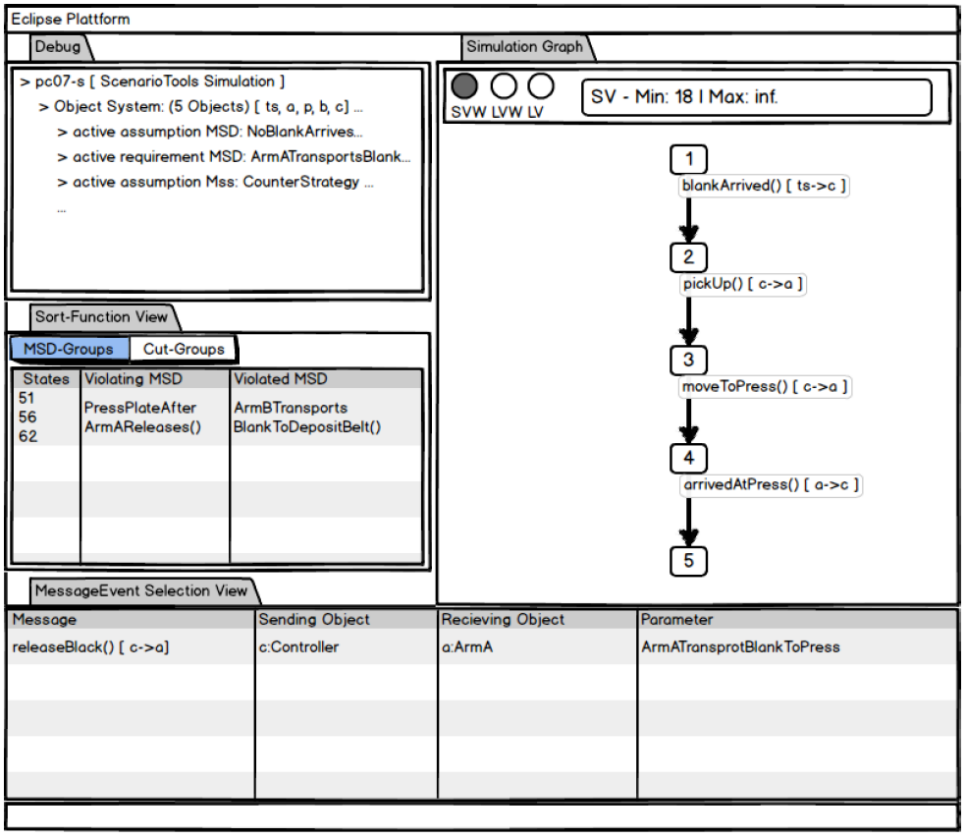


Abbildung 4.1: MockUp inkl. zusätzlicher Informationen und Funktionen

indem ein erreichter Zustand unmittelbar vor einer Verletzung im History-graph seine Gruppenzugehörigkeit anzeigt.

Kapitel 5

Verwandte Arbeiten

5.1 UPPAAL TIGA und PlayGo

UppaalTiga arbeitet mit LCS (Live Sequence Charts), einer Vorgängerversion von MSDs, an der Analyse szenariobasierter Spezifikationen. Dabei werden Timed-Game-Automaten verwendet, an Hand derer man ebenfalls ein Counter-Play-Out durchspielen kann [1]. Greenyer sagt über die Aussagekraft deren Synthese folgendes aus.

> „Environment events that are not enabled in any LSC will not occur. The latter assumption is overly optimistic, and has no practical justification.“
> - [11, Kap. 8.2]

Darüber hinaus wird hier noch keine weitere Unterstützung angeboten, gefundene Fehler in einer Spezifikation effektiv zu beheben.

Maoz und Sa'ar haben in PlayGo ein Counter-Play-Out entwickelt, bei dem man den Systemverlauf entlang der Lebenslinien in aktiven LCSs verfolgt [17] [19]. Maoz und Sa'ar merken darüber hinaus an, dass sich zwar durchaus mit der Synthese von Steuerungen durch Szenarien beschäftigt wird, über die Behebung von Fehlern in solchen Spezifikationen jedoch bisher wenig veröffentlicht wurde.

5.2 Formale Fehler Modellierung

Breitling modelliert in seiner Dissertation [3] verschiedene Fehler, die bei verteilten Systemen auftreten können. Die Dissertation stellt Methoden vor, ein bestehendes System zu Modifizieren und somit hinsichtlich nicht berücksichtigter Ereignisse in der Umgebung Robuster zu machen. Im Fokus stehen hierbei hauptsächlich Fehler bei der Datenübertragung, Ausfälle oder andere Störungen, die nicht in einer Systemspezifikation abgefangen werden. Dabei stellt er auch formal vor, wie dem Zustandsgraphen Transitionen

hinzugefügt werden können und somit dem System ein „Ausweg“ geboten wird. Eine praktische Hinführung zu Fehlern wird hier jedoch auch nicht vorgestellt.

Kapitel 6

Résumé

6.1 Zusammenfassung

Ziel dieser Bachelorarbeit war es das Counter-Play-Out für fehlerhafte Spezifikationen von reaktiven Systemen zu erweitern. Die Erweiterungen sollen dem Entwickler helfen, schnell einen Überblick über die möglichen Fehler zu bekommen. Außerdem soll die Orientierung bis zu etwaigen Fehlerzuständen verbessert werden.

Hierzu wurden neue Konzepte erarbeitet, die noch einmal kurz erläutert werden. Durch drei Flags an jedem Knoten des Graphen der Counterstrategie wird der vom jeweiligen Knoten noch erreichbare Teilgraph charakterisiert. Dabei stehen die Flags für *Sicherheitsverletzung erreichbar*, *Lebendigkeitsverletzung erreichbar* und *bereits in Lebendigkeitsverletzung*. Zudem gibt es Informationen darüber, wie weit Sicherheitsverletzungen vom aktuellen Knoten entfernt sind. Mit Hilfe einer Gruppierungsfunktion wird dem Entwickler aufgezeigt, in wie weit sich die Fehlerzustände, die im Counter-Play-Out erreichbar sind, ähneln. Diese Ähnlichkeit, die verschiedene Fehlerzustände einer Gruppe zuordnet, lässt direkt Rückschlüsse über die Fehlerursache zu. Somit helfen die Gruppen dem Entwickler die relevanten Informationen aus dem Counter-Play-Out zu ziehen, wodurch die Fehler besser verstanden und somit behoben werden können. Allerdings muss noch konzipiert werden, wie ein direkter Bezug von den gruppierten Zuständen zu dem Play-Out-Graphen hergestellt werden kann.

Von den vorgeschlagenen Orientierungshilfen aus Tabelle 1.1 konnten folgende nicht mehr vertieft werden: die Anzahl von Fehlern, die noch erreicht werden können und eine Sprungfunktion kurz vor einen Fehlerzustand. Diese Sprungfunktion eignet sich allerdings dafür, oben genannten Bezug zwischen den gruppierten Zuständen und dem Play-Out-Graphen herzustellen. Grundsätzlich führen die vorgestellten Konzepte bereits zu mehr Übersichtlichkeit, einer besseren Orientierung und somit zu einer

Zeitersparnis im Korrekturprozess einer Spezifizierung.

6.2 Ausblick

Es kann passieren, das durch das Counter-Play-Out ein Fehler verstanden wird, es jedoch keine gewünschte Anforderung oder realistische Annahme gibt, die diesen beheben kann. In dieser Situation möchte man evtl. Ausnahmen definieren. Breitling stellt in seiner Dissertation [3] Modifikationen vor, die vergleichbar mit Exeptionhandling sind. Im Kapitel 5 fügt er dem Zustandsgraphen eines reaktiven Systems Transitionen hinzu. Diese Transitionen führen von jedem Zustand durch ein definierten Fehlerereignis in einen *Fangzustand*. Diese Idee kann man auf das Thema dieser Arbeit übertragen. Angenommen der Entwickler muss einen Fehler beheben, der durch eine Folge von Ereignissen auftreten kann, die als Störung zu werten ist. Dann soll diese nicht Teil einer Spezifikation sein. Hier könnte, je nach Möglichkeit des Systems ein Neustart, Notaus, Alarm etc. angedeutet werden.

Wurde eine Spezifikation derart geändert, dass diese nun realisierbar ist, bleibt die Frage, ob die vorgenommenen Änderungen auch realistische Szenarien abbilden. An dieser Stelle könnte noch eine Kontrollfunktion angeschlossen werden, die alle Änderungen aufzeigt. Gordon und Harel [10] stellen ein Konzept vor, wie aus natürlicher Sprache ausführbare Szenarien generiert werden können. Analog zu dieser Arbeit könnten die neuen oder veränderten Szenarien zur Kontrolle in natürliche Sprache rückübersetzt werden und dadurch auch von jemandem kontrolliert werden, der nicht mit der Semantik der MSDs vertraut ist.

Anhang A

Alle Systemzustände der Beispielspezifikation

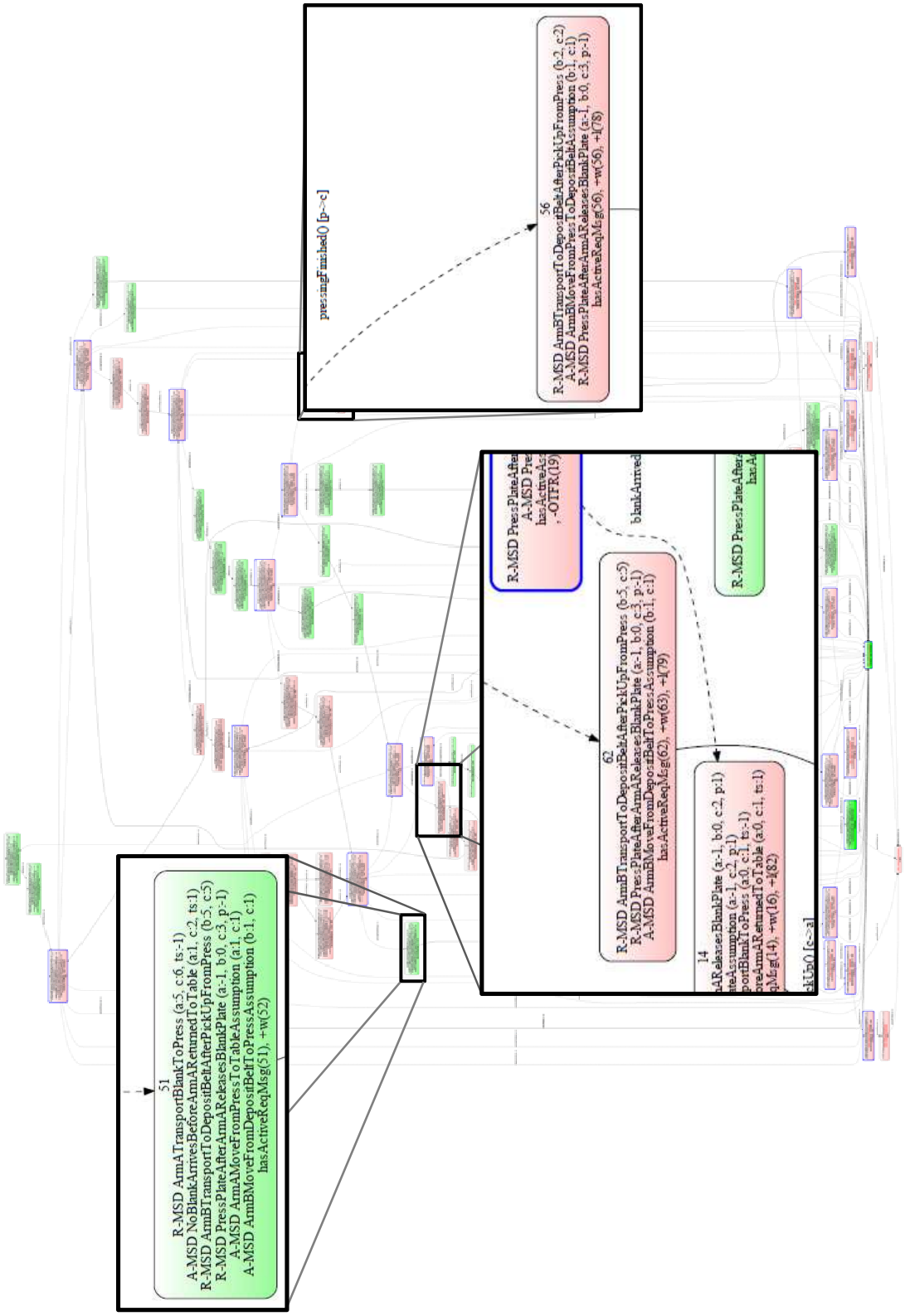


Abbildung A.1: Systemzustände

Anhang B

Counterstrategie der Beispielspezifikation

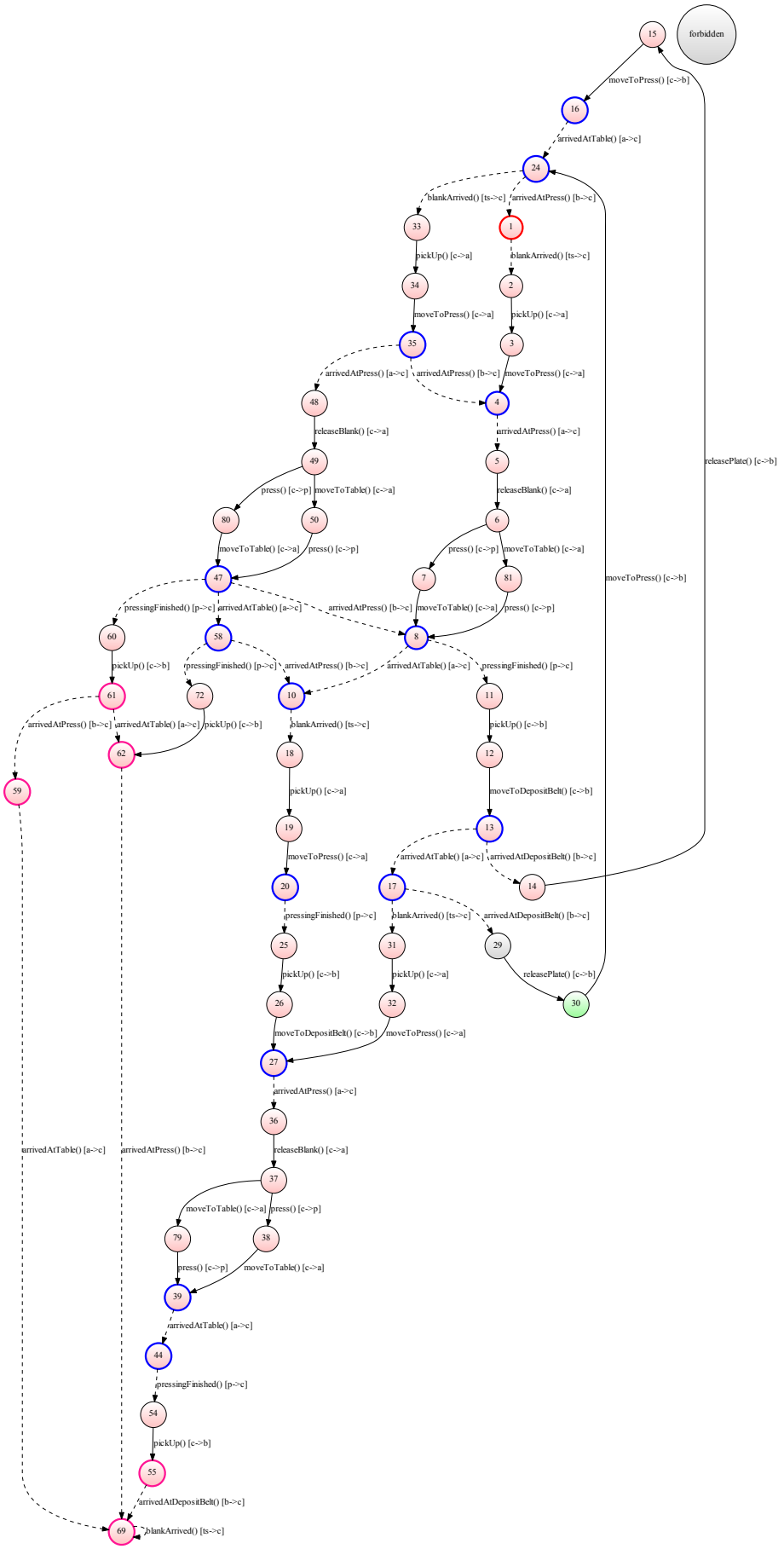


Abbildung B.1: Counterstrategie

Anhang C

Counter-Play-Out der Beispielspezifikation

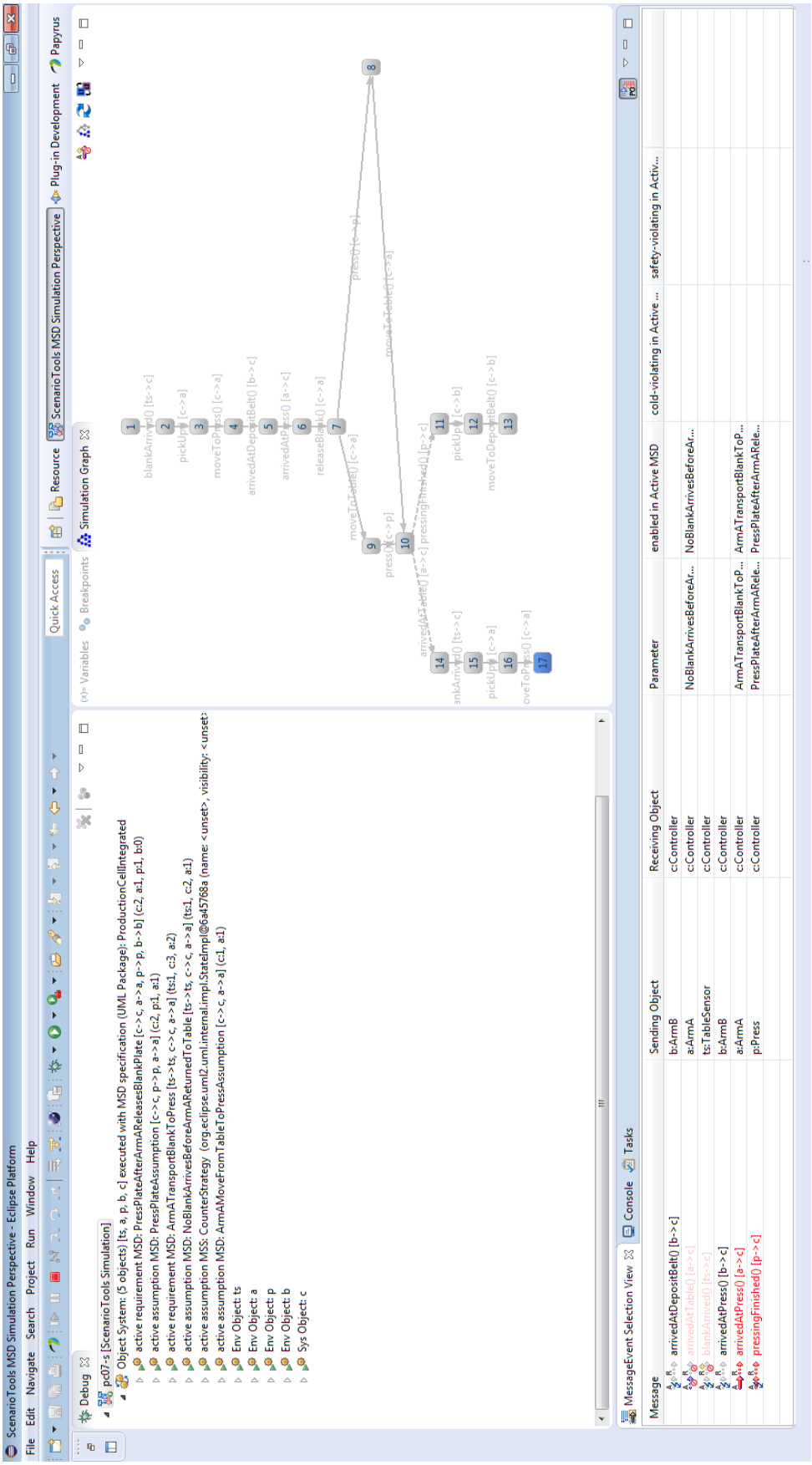


Abbildung C.1: Screenshot des Counter-Play-Outs

Abbildungsverzeichnis

1.1 Von Systemspezifikation zu Systemsteuerung 2

2.1 Beschreibung von System und dessen Umgebung 5
2.2 Skizze einer Produktionszelle, einem reaktiven System 6
2.3 Anforderungen R1-R3 aus Tab. 2.1 10
2.4 Annahmen A1-A3 von insg. 6, wobei A4-A6 der 3. sehr ähnlich sind 11
2.5 Schritte zur Fehleridentifizierung vgl. Anhänge A-C 13
2.6 MessageEvent Selection View 13

4.1 MockUp inkl. zusätzlicher Informationen und Funktionen . . 22

A.1 Systemzustände 28

B.1 Counterstrategie 30

C.1 Screenshot des Counter-Play-Outs 32

Tabellenverzeichnis

1.1 Mögliche Orientierungshilfen im Counter-Play-Out 3

2.1 Anforderungen an die Produktionszelle aus Abb. 2.2 8
2.2 Semantik der Nachrichten in MSDs 9
2.3 Anforderung 2 wird in MSD Semantik übertragen 11

3.1 Kombinationsmöglichkeiten 16
3.2 Vergleich der Zustände unmittelbar vor einer Verletzung . . . 18

Literaturverzeichnis

[1] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer Berlin Heidelberg, 2007.

[2] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62:139–169, 2004.

[3] M. Breitling. *Formale Fehlermodellierung für verteilte reaktive Systeme.* PhD thesis, Technical Univertity of Munich, 2001.

[4] C. Brenner, J. Greenyer, J. Holtmann, G. Liebel, G. Stieglbauer, and M. Tichy. 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)*, volume 67. EASST, 2014.

[5] C. Brenner, J. Greenyer, and V. Panzica La Manna. 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)*, volume 58. EASST, 2013.

[6] F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg, 2005.

[7] K. Chatterjee, T. A. Henzinger, and N. Piterman. Algorithms for büchi games. *arXiv preprint arXiv:0805.2620*, 2008.

[8] A. Cockburn. *Writing effective use cases.* Addison-Wesley, 2001.

[9] A. David et al. Uppaal Tiga – Uppaal for Timed Games, v0.16. `http://people.cs.aau.dk/~adavid/tiga/`. letzer Zugriff: Juni 2013.

[10] M. Gordon and D. Harel. Generating executable scenarios from natural language. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 5449 of *Lecture Notes in Computer Science*, pages 456–467. Springer Berlin Heidelberg, 2009.

[11] J. Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, Paderborn, October 2011.

[12] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 433–443, New York, NY, USA, 2013. ACM.

[13] J. Greenyer and J. Frieben. Consistency checking scenario-based specifications of dynamic systems by combining simulation and synthesis. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 2:1–2:9, New York, NY, USA, 2012. ACM.

[14] T. Gutjahr. Scenariotools counter-play-out - simulation zur analyse von unrealisierbaren szenariobasierten spezifikationen. Bachelor's thesis, Lebniz University of Hanover, 2014.

[15] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.

[16] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[17] D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling*, 2(2):82–107, 2003.

[18] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Berlin Heidelberg, 1985.

[19] S. Maoz and Y. Sa'ar. Counter play-out: Executing unrealizable scenario-based specifications. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 242–251, Piscataway, NJ, USA, 2013. IEEE Press.

[20] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.