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

ScenarioTools Counter-Play-Out Simulation zur Analyse von unrealisierbaren szenariobasierten Spezifikationen

Bachelorarbeit

im Studiengang Informatik

von

Timo Gutjahr

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

Hannover, 04.03.2014

Inhaltsverzeichnis

1. Einleitung

Wenn wir uns heute umschauen, dann können wir überall softwaregesteuerte Systeme entdecken. Oft sind es nur kleine Systeme. Allerdings gibt es mittlerweile eine beträchtliche Anzahl an komplexen Systemen, die ein großes Spektrum an Funktionen abdecken. Diese Systeme bestehen aus vielen Komponenten, die im Zusammenspiel miteinander viele Aufgaben und diese zum Teil sogar gleichzeitig ausführen. Die einzelnen Komponenten reagieren dabei ständig auf Ereignisse aus ihrer Umwelt und stimmen sich mit dieser ab. Daher werden derartige Systeme als *reaktive Systeme* bezeichnet.

Produktionsroboter und sich autonom bewegende Fahrzeuge fallen unter diese Kategorie. So bewegen sich beispielsweise kleine Transporter autonom in Lagerhallen, um Waren zu befördern. Gleichfalls autonom bewegen sich Autos, die sich auf der Autobahn in Fahrzeugkolonen einfädeln und anschließend die Steuerung übernehmen. Auch Fertigungsanlagen, die in große Fabrikhallen stehen, gehören zu diesen Systemen.

Typischerweise liegen die Anforderungen an solche Systeme in textuellen Spezifikationen vor. Darin sind die Funktionen und das Verhalten des Systems mit verschiedenen Use Cases beschrieben. Diese wiederum können aus mehreren Szenarien bestehen, wobei die Spezifikationen oft sehr umfangreich sind und es sich daher nicht mehr ohne Weiteres prüfen lässt, ob sie frei von Widersprüchen sind oder kurz: *Realisierbar* sind. Dabei liegt ein Widerspruch in zwei Szenarien genau dann vor, wenn die Szenarien die gleiche Situation mit widersprüchlichen Anforderungen beschreiben.

Beispielsweise könnte durch folgende zwei Szenarien ein Widerspruch bei autonomen Fahrzeugen entstehen: *1. Wenn ein Fahrzeug von rechts kommt, musst angehalten werden. 2. Wenn auf einer Vorfahrtsstraße gefahren wird, und ein Fahrzeug von rechts kommt, kann weitergefahren werden.* Es wird angenommen, dass die Szenarien immer dann aktiviert werden, wenn ihre erste Bedingung erfüllt ist. Wenn das Fahrzeug auf der Vorfahrtstraße unterwegs ist, dann ist Szenario 2 aktiv. Kommt nun ein Fahrzeug von rechts, sind sowohl die zweite Bedingung in Szenario 2 als auch die erste Bedingung in Szenario 1 erfüllt, sodass Szenario 1 ebenfalls aktiviert wird. Im nächsten Schritt fordern die Szenarien unterschiedliche Aktionen (Anhalten und Weiterfahren), wodurch ein Widerspruch entsteht. Dies macht die Spezifikation unrealisierbar.

Um an dieser Stelle Analyseverfahren zu unterstützen, wurden bereits szenariobasierte Modellierungsmethoden entwickelt deren Ziel es ist, Szenarien aus textuellen Spezifikation intuitiv in formale Spezifikationen umzuformen. Auf dieser Basis lässt sich bestimmen ob Spezifikationen Widersprüche enthalten. In *ScenarioTools* ist dies auf Basis von *MSD Spezifikationen* umgesetzt.

ScenarioTools kann diese Spezifikationen in einer Simulation ausführbar machen. Zudem kann durch eine *Synthese* die Spezifikation auf Widersprüche geprüft werden. Enthält die Spezifikation keine Widersprüche, wird eine Strategie erzeugt, die das System steuern

kann. Ist dies nicht der Fall, wird eine *Gegenstrategie* erzeugt. Mittels dieser Gegenstrategie liegt eine Beschreibung vor, wie die Umwelt das System in eine Situation bringen kann, in der es die Spezifikationen verletzt. Die Gegenstrategie kann dabei als Graph betrachtet werden, der allerdings bei etwas umfangreicheren Spezifikationen nicht einfach zu interpretieren ist. Aus diesem Grund soll die Analyse von unrealisierbaren Spezifikationen verbessert werden. Dazu wird die Gegenstrategie in die Simulation integriert.

Die Ausführung einer Simulation kombiniert mit einer Gegenstrategie wird *Counter-Play-Out* (CPO) genannt. Mit dem CPO soll der Benutzer auf eine spielerische Art und Weise bei der Analyse von szenariobasierten Spezifikationen unterstützt werden. Dabei steuert er die Aktionen des Systems und versucht die Anforderungen zu erfüllen. Die Simulation übernimmt die Steuerung der Umwelt und beginnt ihrerseits die Gegenstrategie umzusetzen und damit die Anforderungen zu verletzen. Allerdings ist dies kein Spiel, das der Benutzer gegen die Umwelt gewinnen kann, aber genau darin liegt der Erkenntnisgewinn: Wenn der Benutzer im CPO erkennt, in welchen Situationen seine Spezifikation Widersprüche aufweist, kann er diese gezielt beheben.

Um das CPO in ScenarioTools umzusetzen, habe ich die Gegenstrategie für die Simulation ausführbar gemacht. Dazu habe ich die Gegenstrategie mit Hilfe einer TGG Transformation in das Format der MSD Spezifikation abgebildet. Anschließend habe ich die Simulation so erweitert, dass sie die Gegenstrategie zusammen mit den MSD Spezifikationen ausführen kann. Auf diese Weise kann auch eine Strategie des Systems simuliert werden.

Die vorliegende Arbeit beschäftigt sich mit genau dieser Problematik. Dazu werde ich zunächst die technischen Grundlagen in Kapitel 2 erläutert. Eine Anforderungsanalyse führe ich in Kapitel 3 durch. Meinen Entwurf für eine Lösung stelle ich in Kapitel 4 vor. Kapitel 5 enthält Informationen über die Implementierung. Verwandte Arbeiten sind in Kapitel 6 zusammengefasst. In Kapitel 7 folgt die Zusammenfassung meiner Arbeit.

2. Grundlagen

In diesem Kapitel möchte ich die für meine Umsetzung des Counter-Play-Out nötigen Grundlagen vermitteln. In 2.1 erkläre ich wie mit Hilfe von MSD Spezifikationen reaktive Systeme beschrieben werden können. Anschließend stelle ich in 2.2 ScenarioTools vor, die Entwicklungsumgebung, mit der MSD Spezifikationen modelliert werden können und in der ich meine Lösung implementiert habe. Im letzten Abschnitt (Kapitel 2.3) erkläre ich die Funktionsweise von TGGs. Hierbei handelt es sich um eine Modelltransformationssprache, die bei der Implementation von Counter-Play-Out eine zentrale Rolle übernimmt.

2.1. MSD Spezifikation

MSD Spezifikationen sind ein Ansatz um Systeme und ihr Verhalten mit Hilfe von Szenarien zu beschrieben. Im Fokus stehen reaktive Systeme, also Systeme die mit ihrer Umwelt interagieren. Dabei kann beschrieben werden, was ein System tun kann, muss bzw. nicht tun darf. Harel und Marelly erklären den Vorteil von szenariobasierten Beschreibung von Systemen in [Come, Let's Play] damit, dass es intuitiver darzustellen ist, wie ein System in einem bestimmten Szenario reagiert und welche anderen Komponenten dabei beeinflusst werden. Welche Szenarien ein System erfüllen soll und wie das System dabei mit der Umwelt interagiert, ist oft schon zu Beginn des Modellierungsprozesses bekannt. Deutlich schwieriger ist es allerdings diese Szenarien erst zu analysieren, um anschließend für jede Komponente einzeln zu definieren, auf welche Ereignisse es reagieren soll und in welchem Zustand es sich dafür befinden muss.

MSDs sind eine Erweiterung des in [Come, Let's Play] beschriebenen Ansatzes. Dort werden szenariobasierte Spezifikationen mit Live Sequence Charts (LSC) modelliert. Dabei handelt es sich um eine Erweiterung von UML Sequenzdiagrammen, bei denen Nachrichten in verschiedene Kategorien unterteilt werden können. Es gibt Nachrichten, die geschehen müssen, welche die geschehen können und die, die nicht geschehen dürfen. In dieser Arbeit werden MSD Spezifikation in der Art verwendet, wie sie in der Dissertation von Greenyer [G11] mit den aktuellen Erweiterungen [BGP13] [BGPGS13] definiert sind.

2.1.1. Einführung das Beispiels (Teil 1)

Das Beispiel behandelt eine Produktionszelle wie sie beispielsweise in einer Fabrik zum Einsatz kommt. Es ist den ScenarioTools Quellen entnommen. In Abbildung 2.1 ist eine

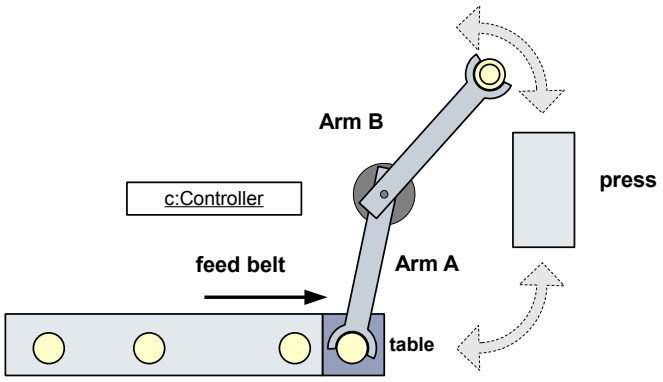


Abbildung 2.1.: Aufbau einer Produktionszelle in Anlehnung an [G13]

Produktionszelle zu sehen. Sie besteht aus einem Roboter mit zwei Armen, die von einem Controller gesteuert werden. Hinzu kommen noch ein Fließband sowie eine Presse. Über das Fließband gelangen Rohlinge zu einem Tisch. Von dort aus sollen sie vom Roboter in die Presse gelegt werden und anschließend wieder heraus genommen werden. Der Tisch besitzt einen Sensor, der dem Controller meldet, wenn ein Rohling ankommt. Die Roboterarme haben Sensoren, die signalisieren, dass der Arm am Ziel angekommen ist. Wenn die Presse mit dem Pressen fertig ist, schickt sie eine Nachricht zum Controller. Im Folgenden sind die Anforderungen an das System als informale Szenarien beschrieben:

Anforderung 1: Nachdem ein Rohling am Tisch angekommen ist, muss ArmA ihn vom Tisch nehmen und sich in Richtung Presse bewegen. Sobald er dort angekommen ist, muss er den Rohling in die Presse legen und bewegt sich in Richtung Tisch zurück, wo er dann irgendwann ankommt.

Anforderung 2: Nachdem ArmA einen Rohling in die Presse gelegt hat, muss die Presse pressen. Sobald die Presse damit fertig ist, muss ArmB den Rohling aus der Presse nehmen.

Annahme 1: Wenn sich ArmA in Richtung Presse bewegen soll, kommt er auch irgendwann dort an.

Annahme 2: Wenn sich ArmA in Richtung Tisch bewegen soll, kommt er auch irgendwann dort an.

Annahme 3: Wenn ein Rohling am Tisch angekommen ist, darf der nächste Rohling erst dann ankommen, wenn ArmA wieder zurück am Tisch ist.

2.1.2. MSD Grundlagen

Eine MSD Spezifikation ist eine graphische Methode um das Verhalten von Software und seiner einzelnen Komponenten im Zusammenspiel mit seiner Umwelt zu beschreiben. Dabei besteht eine MSD Spezifikation aus einer Menge von *Modal Sequence Diagrams (MSDs)* und *Modal State Structures (MSSs)*. Mit diesen Diagrammen lässt sich

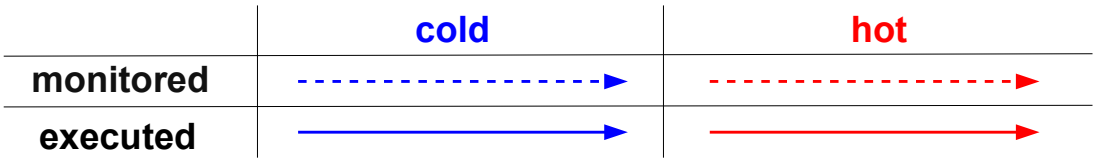


Abbildung 2.2.: Die verschiedenen Nachrichtenarten. cold - darf verletzt werden, hot darf nicht verletzt werden. monitored - kann geschehen, executed - muss geschehen. In Anlehnung an [G13]

beschreiben welche Ereignisse passieren dürfen, welche passieren müssen und welche Ereignisse nicht auftreten dürfen. Zudem lässt sich eine Einteilung in schwerwiegende Fehler und nicht schlimme Fehler treffen. Auch das Verhalten der Umwelt kann mit MSD Spezifikationen berücksichtigt werden.

Objekte, die in einer MSD Spezifikation modelliert werden, gehören zu einem *Objektsystem*. Darin lassen sich die Objekte in zwei Klassen unterteilen. Eine Klasse bildet das *Umweltsystem*, dem alle Objekte zugeordnet sind, die zur Umwelt gehören. Aus diesem Grund wird es auch vereinfacht *Umwelt* genannt. Dem gegenüber steht das *System*, welches alle Objekte des Systems beinhaltet. Die *Nachrichten*, die vom System verschickt werden, sind *kontrollierbare* Nachrichten. Die Nachrichten, welche die Umwelt verschickt, sind hingegen *unkontrollierbar*. Bezogen auf das Produktionszellen-Beispiel stellt der Controller das System dar, die weiteren Komponenten werden von ihm gesteuert und sind Teil der Umwelt. Über die Nachrichten, die der Controller verschickt, hat er volle Kontrolle. Auf das Eintreffen von Nachrichten aus seiner Umwelt muss der Controller warten, sie sind für ihn unkontrollierbar.

Modal Seqence Diagrams beschreiben das Verhalten durch sequenzielle Abläufe. Dabei werden Objekte als senkrechte, gestrichelte Linien dargestellt, an deren oberem Ende ihr Name und ihr Typ stehen. Diese Linien werden *Lebenslinien* genannt. In den folgenden Abbildungen stehen die Namen der Komponenten, die zum System gehören in Rechtecken, die die zur Umwelt gehören haben einen wolkenähnlichen Rahmen. Zwischen ihnen können Nachrichten ausgetauscht werden.

Nachrichten werden von einer Systemkomponente gesendet und von einer anderen empfangen. Zudem haben Nachrichten einen Namen. Dieser entspricht der Funktion, die bei der empfangenen Komponente ausgelöst werden soll. Hinzu kann noch ein Parameter vom Typ *Integer*, *String* oder *Boolean* kommen. Nachrichten, die durch einen durchgezogenen Pfeil dargestellt sind, müssen auftreten und heißen *executed*. Gestrichelt gezeichnete Nachrichten können auftreten und werden als *monitored* bezeichnet. Nachrichten, die blau gezeichnet sind, können verletzt werden. Diese Nachrichten werden *cold* genannt. Hingegen rot gezeichnete Nachrichten dürfen nicht verletzt werden. Sie sind *hot*. Eine grafische Zuordnung der Nachrichten und ihrer Bezeichner ist in Abbildung 2.2 zu sehen.

Während der Ausführung treten eine Reihe von unendlich vielen Nachrichten auf. Dabei reagiert das System auf Umweltnachrichten mit einer Sequenz an Systemnachrichten. Wenn diese Nachrichten auf Nachrichten in MSDs abgebildet werden, dann müssen

folgende Eigenschaften übereinstimmen, wobei dieser Vorgang in diesem Kontext *unifizieren* genannt wird. Dabei muss der Name der Nachrichten übereinstimmen und der Parameter vom gleichen Typ sein. Zudem muss die sendende und die empfangende Komponente der Nachricht zur Lebenslinie, von der die Nachricht ausgeht bzw. ankommt, passen. Wird die erste Nachricht in einem MSD mit einer auftretenden Nachricht unifiziert, wird es *aktiviert*. Allerdings kann es zu jedem MSD während der Laufzeit nur ein aktives MSD geben. Die Menge aller aktiven MSDs bestimmen den Zustand des Systems.

Wird ein MSD ausgeführt, zeigt der *Cut* an in welchem Zustand sich das MSD befindet. Der Cut wird durch eine gepunktete waagerechte Linie dargestellt, die zwischen den Nachrichten, die bereits geschehen sind, und jenen die noch nicht geschehen sind, verläuft. Der Cut ist executed, wenn die folgende Nachricht executed ist. Anderenfalls ist er monitored. Ist die folgende Nachricht hot, so ist auch der Cut hot. Enstprechend im Fall cold.

Die Nachrichten, die auf den Cut folgen, sind erlaubt. Erlaubte Nachrichten, die executet sind, werden als *aktive Nachrichten* bezeichnet. Befindet sich keine Nachricht vor dem Cut, kann das aktive MSD terminiert werden. Tritt eine Nachricht auf, wird in allen aktiven MSDs geschaut, ob die Nachricht mit den erlaubt Nachrichten unifizierbar ist. Trifft dies auf eine erlaubte Nachricht zu, wird sie ausgeführt und der Cut weiter gesetzt. In allen aktiven MSDs, in denen die Nachricht stattdessen mit einer Nachricht, die nicht erlaubt ist, unifiziert werden kann, kommt es zu einer Verletzung. Ist der Cut dabei cold, führt die Verletzung zur Terminierung des aktiven MSD. Hier wird von einer kalten Verletzung (*cold violation*) gesprochen. Ist der Cut jedoch hot, führt die Verletzung zu einer Sicherheitsverletzung (*safety violation*). Eine solche Verletzung darf niemals geschehen und hat die Terminierung alles MSDs zur Folge. Befindet sich der Cut vor einer Nachricht, die executed ist und nicht mehr auftreten kann, kommt es zu einer Lebendigkeitsverletzung (*liveness violation*). Das bedeutet, dass eine Aktion zugleich passieren muss und niemals geschehen kann.

2.1.3. Beispiel Teil 2

Im Folgenden wird die Idee von MSDs an Hand des Beispiels erläutert. In Abbildung 2.3 ist auf der linken Seite die erste Anforderung als MSD dargestellt. Nach dem Ankommen eines Rohlings soll der Roboter diesen aufnehmen, zur Presse bringen und anschließend zurückkehren. Dabei ist das textuell beschriebene Szenario eins zu eins in das MSD übersetzt worden.

Bei genauem Hinschauen fällt auf, dass die Szenarien nicht ausschließlich informal sind. So induziert jedes *muss*, dass eine Aktion executed sein soll und jedes *irgendwann* oder *sobald* eine monitored Nachricht.

Die erste Nachricht in einem MSD ist für gewöhnlich immer cold und monitored. Sie dient zur Initialisierung des MSD. Wurde das MSD initialisiert, befindet sich der Cut hinter der Nachricht blankArrived() und vor pickUp(). Damit ist die Nachricht pickUp()

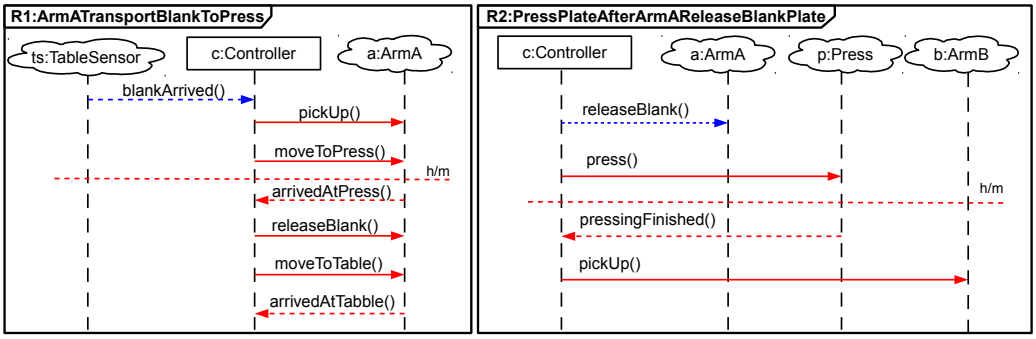


Abbildung 2.3.: Zwei MSDs, die die Anforderungen an die Produktionszelle beschreiben. In Anlehnung an [G13]

erlaubt. Da die Nachticht hot und executed ist, ist dies auch der Cut. In der Abb. 2.3 ist auf der rechten Seite die zweite Anforderung an das System dargestellt. Mit diesem MSD wird beschrieben, dass die Presse pressen soll sobald ein Rohling hinein gelegt wurde. Anschließend soll ArmB den Rohling aufnehmen.

2.1.4. MSD Erweiterungen

Im Folgenden Abschnitt werden einige Erweiterungen von MSDs erklärt, mit denen sich das Verhalten von Systemen noch präziser darstellen lässt. Beginnend wird gezeigt wie sich Verbote modellieren lassen. Anschließend geht es um dynamische Objektbindung und die Möglichkeit Verhalten zustandsbasiert darzustellen. Zum Ende des Abschnitts wird beschrieben wie sich das Verhalten der Umwelt einschränken lässt.

Mit Hilfe von *Verbot-Fragmenten* bietet sich eine Möglichkeit Verhalten, das während eines Szenarios nicht auftreten darf, zu beschreiben. Fragmente werden dabei durch einen Kasten dargestellt, der die Lebenslinien umschließt, die von der Bedingung betroffen sind. Zusätzlich sind sie mit der Abkürzung *neg* gekennzeichnet.

Verbot-Fragmente beinhalten Nachrichten, die nicht auftreten dürfen. Auf diese Weise kann verhindert werden, dass eine Nachricht eintritt solange der Cut nicht an dem Fragment vorbei ist. Tritt eine solche Nachricht doch auf, führt dies zu einer Verletzung. Welcher Art die Verletzung ist, wird dabei durch die Temperatur der verbotenen Nachricht bestimmt, nicht durch den Cut.

Bislang wurde davon ausgegangen, dass zum Zeitpunkt der Aktivierung eines MSDs feststeht, welche Komponenten, im Folgenden als Objekte bezeichnet, aus dem Objektsystem zu welchen Lebenslinien gebunden werden. Diese Art der Zuordnung wird *konkrete Objektbindung* genannt. In den MSDs ist diese Eigenschaft durch unterstrichene Objektnamen der Lebenslinien gekennzeichnet. Für die Modellierung dynamischer Systeme kann die konkrete Zuweisung von Objekten zu Lebenslinien zum Zeitpunkt der Aktivierung jedoch unvorteilhaft sein. Bei großen Objektsystemen mit vielen Verände-

rungen können eventuell nicht alle Objektkonstellationen vorausgesagt werden. Damit hier nicht alle möglichen Objekte in Betracht gezogen werden müssen, gibt es *dynamische Objektbindungen*. Dabei werden Objekte erst gebunden sobald sie das erste Mal angesprochen werden.

Zur Modellierung von zustandsabhängigem Verhalten gibt es Modal State Structures (MSS). Sie sind eine erweiterte Form von (Endlichen) Automaten. Sie bestehen aus Zuständen, die wie Nachrichten in den MSDs hot oder cold sein können, sowie entweder executed oder monitored sind. Zustände die hot sind, sind mit einem h gekennzeichnet und haben einen roten Rahmen. Mit einem c und einem blauen Rahmen werden Zustände, die cold sind, gekennzeichnet. Monitored Zustände haben einen doppelten Rahmen, executed Zustände einen einfachen Rahmen. Ein MSS besitzt genau einen Startzustand jedoch keinen expliziten Endzustand. Es kann unendlich lange aktiv sein. Die verschiedenen Zustände werden durch Pfeile mit Nachrichten verbunden. Jede Nachricht hat jeweils ein empfangendes und ein sendendes Objekt sowie einen Namen, welcher der Funktion entspricht, die es aufrufen soll. Der *aktuelle Zustand* eines MSS wird durch einen Zustand beschrieben, der sich ähnlich verhält wie der Cut in einem MSD. Ein aktueller Zustand entspricht immer genau dem Zustand, in dem sich das MSS befindet. Ist der Zustand hot, ist auch der aktuelle Zustand hot. Anderenfalls ist er cold. In einem MSS sind alle Nachrichten erlaubt, die ihren Ursprung im aktuellen Zustand haben. Tritt eine Nachricht auf, die nicht erlaubt ist jedoch mit einer anderen Nachricht im MSS unifiziert werden kann, kommt es zu einer Verletzung. Ist der aktuelle Zustand hot, ist dies eine Sicherheitsverletzung. Ist der aktuelle Zustand hingegen cold, wird das aktive MSS beendet. Nachrichten die nicht im MSS definiert sind, werden ignoriert.

Der Grundgedanke von reaktiven Systemen schließt mit ein, dass ein System immer auf die Aktionen der Umwelt reagieren kann. Dabei ist die Umwelt nicht vom System kontrollierbar und das System muss auf alle möglichen Umweltnachrichten reagieren können. Dies hat zur Folge, dass sehr viele Szenarien modelliert werden müssen die niemals auftreten werden. Da jedoch in den meisten Fällen Informationen über die Umwelt vorliegen, die Aktionen der Umwelt einschränken (wie es z.B. physische Eigenschaften oder Abhängigkeiten tun), können Entwickler *Annahmen* über das Verhalten der Umwelt treffen. Diese Annahmen können in *Environment Assumption MSDs* bzw. *MSSs* definiert werden. Dabei handelt es sich um MSSs und MSDs die um die Eigenschaft *«Environment Assumption»* erweitert werden, diese Information ist im Namensfeld wiederzufinden. Regeln, die das Verhalten des Systems beschreiben, heißen *Requirement MSDs* bzw. *MSSs*. Durch die getrennte Modellierung von Anforderungen und Annahmen muss nun auch bei Verstößen unterschieden werden. An dieser Stelle lässt sich der spielbasierte Ansatz des *Counter-Play-Outs* erklären. Das System muss die Regeln der Anforderungen befolgen. Die Umwelt muss sich an die Regeln der Annahmen halten. Vorrangiges Ziel des Systems ist es seine Anforderungen zu erfüllen. Im Gegensatz dazu versucht die Umwelt das System so zu lenken, dass es seine Anforderungen nicht einhalten kann. Die Umwelt gewinnt das Spiel wenn sie das System dazu bringen kann die Anforderungen zu verletzten. Damit wäre die Spezifikation *unrealisierbar*. Das System kann das Spiel gewinnen indem es nie gegen seine Anforderungen verstößt oder wenn die Umwelt ihre Annahmen verletzt.

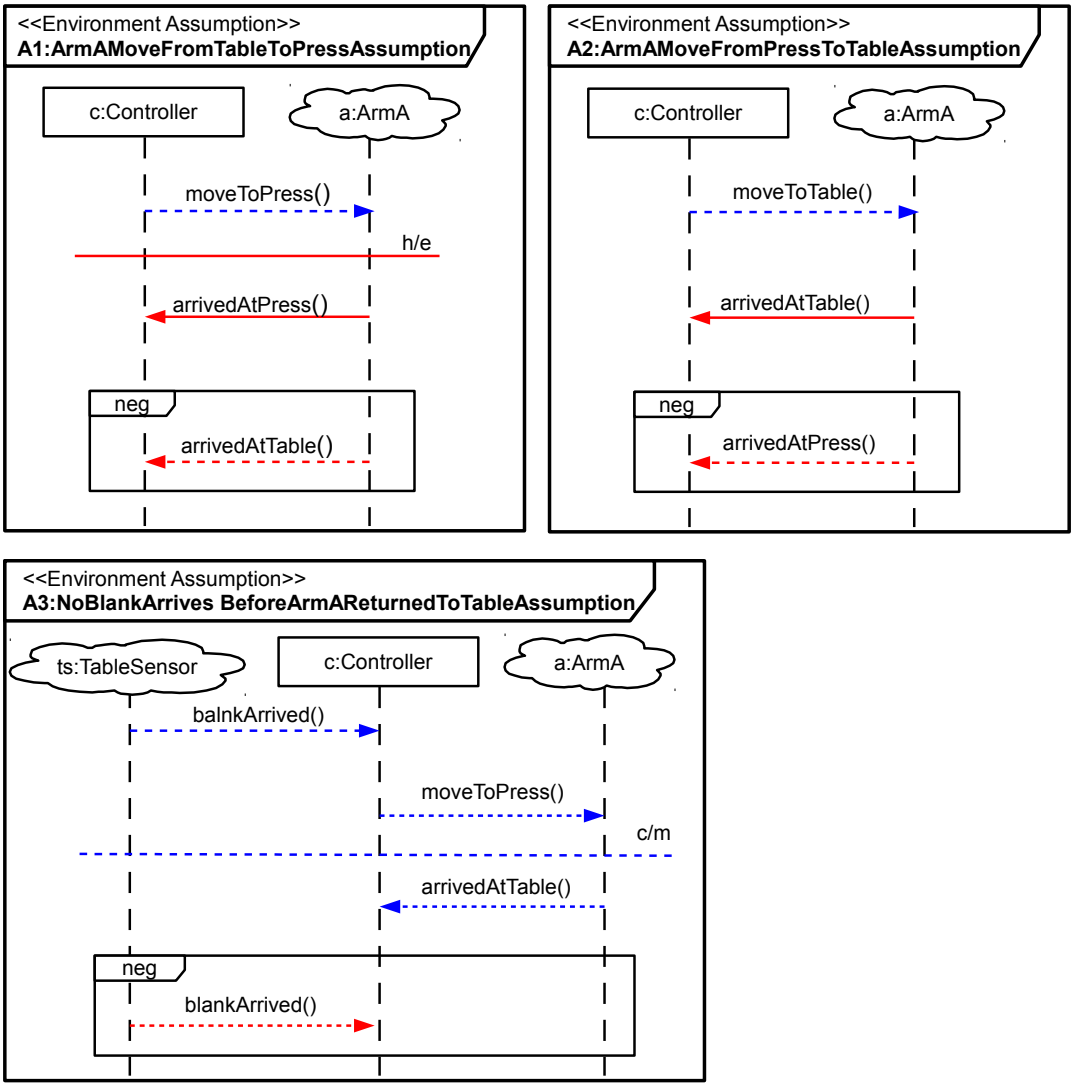


Abbildung 2.4.: Drei Environment Assumption MSDs, die die Annahmen 1-3 der Produktionszelle beschreiben. In Anlehnung an [G13]

2.1.5. Beispiel Teil 3

Im Folgenden wird die Anwendung von Umweltannahmen anhand des Beispiels der Produktionszelle erklärt. Im darauf folgenden Abschnitt ist die Positionsabfolge von ArmA als MSS dargestellt. Anschließend wird verdeutlicht, wie ein Widerspruch in den Spezifikationen entsteht, wie er zu erkennen ist und wie er behoben werden kann.

In Abbildung 2.4 sind die drei Annahmen über das Verhalten der Umwelt zu sehen. Umweltannahme 1 sagt aus, dass ArmA irgendwann bei der Presse ankommt, wenn er sich dorthin bewegen soll. Mit dem Verbot-Fragment wird das „irgendwann“ aus der textuellen Beschreibung noch einmal eingegrenzt. *Annahme 1: Wenn sich Arm A zur Presse*

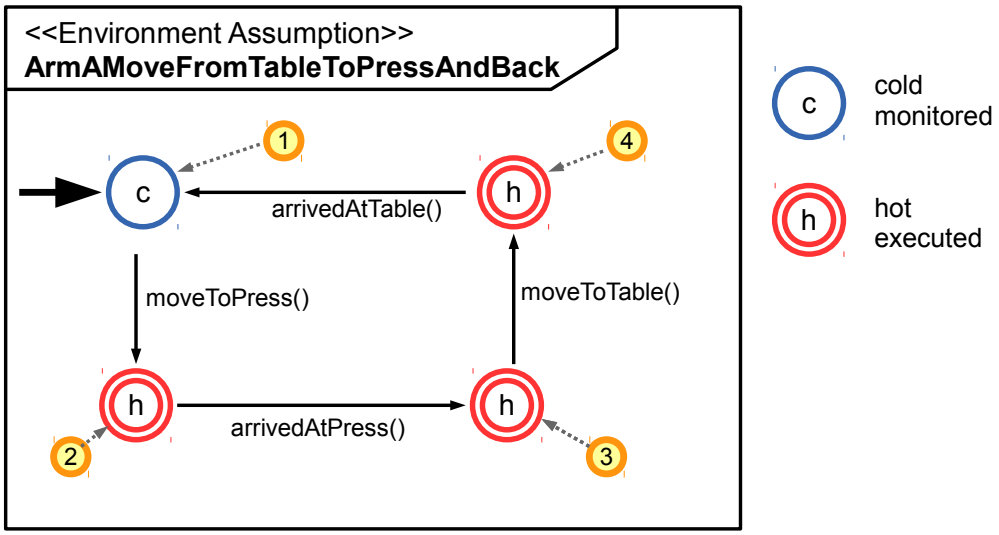


Abbildung 2.5.: Das Environment Assumption MSS beschreibt die Zustände, in denen sich ArmA der Produktionszelle befinden kann.

bewegen soll, dann kommt er dort irgendwann an. Er kommt aber nicht zwischendurch am Tisch vorbei. Tut er dies doch, kommt es zu einer Sicherheitsverletzung, weil die Nachricht in dem Verbot-Fragment hot ist. Umweltannahme 2 beschreibt das Szenario in die andere Richtung: Der Arm bewegt sich von der Presse zum Tisch. Die Annahme ist nicht aktiv. Im dritten Diagramm ist die Ankunft der Rohlinge geregelt: Ein weiterer Rohling darf den Tisch erst erreichen, wenn der Arm den vorherigen Rohling zur Presse gebracht hat und zum Tisch zurückgekehrt ist.

Das MSS ArmAMoveFromTableToPressAndBack in Abbildung 2.5 gibt die Zustände wieder, in denen sich ArmA befinden kann. Dabei startet ArmA in Zustand (1). Hier befindet er sich am Tisch. Triff nun die Nachricht moveToPress() ein, befindet sich der ArmA danach zwischen Tisch und Presse. An dieser Stelle erfüllt das MSS die gleiche Eigenschaft wie die Annahme 1. Würde ArmA als nächstes melden, dass er am Tisch angekommen ist, käme es zur Sicherheitsverletzung, da im aktuellen Zustand lediglich arrivedAtPress() erlaubt ist und arrivedAtTable() an einer anderen Stelle des MSS unifiziert werden kann. Auf gleiche Weise erfüllt das MSS die Annahme 2, wenn es sich im Zustand 4 befindet. Zusätzlich wird mit dem MSS beschrieben, dass sich der Arm immer zwischen Tisch und Presse hin und her bewegt. Sollte diese zusätzliche Eigenschaft nicht mit berücksichtigt werden, so müsste Zustand 3 in einen cold und monitored Zustand geändert und mit Zustand (1) zusammengeführt werden.

Bei der Ausführung von MSD Spezifikation wird davon ausgegangen, dass das System in der Lage ist auf eine Nachricht der Umwelt mit einer Sequenz von Systemnachrichten zu reagieren. Dabei kann es diesen Vorgang schnell genug abschließen bevor es die nächste Umweltnachricht erhält. Bezogen auf die Ausführung bedeutet das konkret, dass sich das System in einem Zustand befindet, in dem es auf eine Aktion der Umwelt wartet. Ist diese Nachricht eingetroffen, werden alle Nachrichten des Systems, die in der jeweiligen

Anforderung erlaubt und executed sind, ausgeführt bis nur noch monitored Nachrichten erlaubt sind. An dieser Stelle geht das System wieder in den Wartemodus.

Im Folgenden wird davon ausgegangen, dass sich das System im folgenden Zustand befindet: ArmA hat den ersten Rohling in die Presse gelegt, den zweiten Rohling aufgenommen und bewegt sich nun in Richtung Presse. Die Presse hat den ersten Pressvorgang gestartet, ihn aber noch nicht beendet. Die Cuts, die in den MSDs R1-2 und A1-3 eingezeichnet sind, beschreiben genau diesen Zustand. Zum jetzigen Zeitpunkt sind nur monitored Nachrichten in den Anforderungen erlaubt. Die Umwelt kann entscheiden welche Nachricht sie ausführt.

Kommt es nun dazu, dass ArmA meldet an der Presse angelangt zu sein (arrivedAtPress()), entsteht ein Widerspruch. Annahme 1 terminiert normal. Der Cut in Anforderung 1 wird eine Nachricht weiter gesetzt. Damit sagt nun Anforderung 1, dass releaseBlank() als nächstes ausgeführt werden muss. Anforderung 2 befindet sich immer noch im Zustand des Wartens auf das Fertig werden der Presse. Würde nun die Nachricht releaseBlank() ausgeführt werden, was laut Anforderungen geschehen muss, führt dies zu einer Sicherheitsverletzung in Anforderung 2: releaseBlank() kann in Anforderung 2 unifiziert werden, wäre zu dem Zeitpunkt jedoch nicht erlaubt.

Um den Widerspruch zu beheben, kann das Verhalten der Umwelt mit weiteren Annahmen weiter eingegrenzt werden. Die erste Idee für eine weitere Annahme ist folgende:

Annahme 4 beta: Wenn die Presse anfängt zu pressen, muss sie diesen Vorgang auch beenden.

Diese Annahme ist allerdings noch nicht ausreichend. Sie sagt zwar aus, dass das Pressen beendet werden muss, jedoch sollte der Pressvorgang von Beginn an betrachtet werden. Daher wird die Annahme ausgeweitet zu:

Annahme 4: Wenn ArmA einen Rohling in die Presse legen und dieser von der Presse gepresst werden soll, dann muss der Pressvorgang danach abgeschlossen werden.

In Abbildung 2.6 ist diese Annahme als MSD zu sehen. Sie sagt aus, dass die Presse mit dem Pressen fertig sein muss, bevor ein neuer Rohling hineingelegt werden kann. Das System befindet sich wieder in der Situation, in der die Presse am Pressen ist und ArmA bereit ist den zweiten Rohling in die Presse zu legen. Die eingezeichneten Cuts stellen diesen Zustand dar. Legt ArmA nun einen Rohling in die Presse, so kommt es zur Verletzung der Annahmen zum Verhalten der Umwelt: Der Cut in Annahme 4 befindet sich vor der Nachricht pressingFinished() und ist damit hot und executed. Die Nachricht releaseBlank() ist in diesem Fall nicht erlaubt. Wodurch eine Sicherheitsverletzung der Annahmen entsteht. Das heißt, die Umwelt tut etwas, das laut Annahmen nicht passieren dürfte. Die daraus resultierende Verletzung der Anforderung in [R2] ist daher durch Fehlverhalten der Umwelt entstanden und steht nicht im Widerspruch zu den Anforderungen.

Daraus lässt sich schließen, dass Environment Assumptions eine sehr mächtige Erweiterung von MSD Spezifikationen darstellen, die nur überlegt eingesetzt werden sollten. Wird jegliches unerwünschte Verhalten der Umwelt verboten, führt dies zwar zur Vermeidung von Widersprüchen in den Anforderungen, ein funktionsfähiges System wird

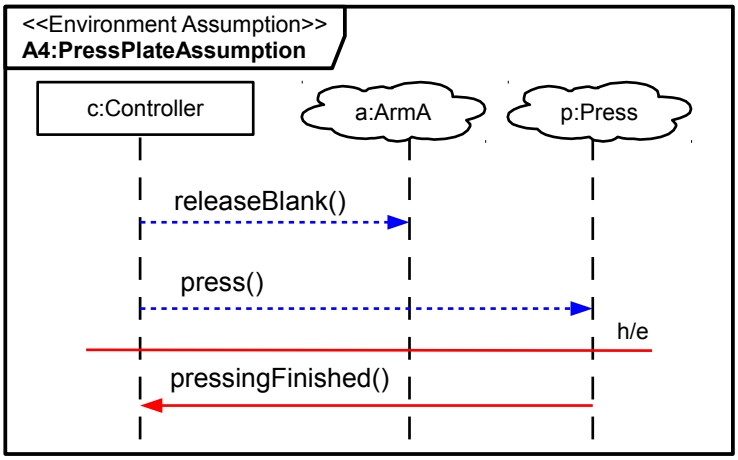


Abbildung 2.6.: Ein Environment Assumption MSD, das die vierte Annahme der Produktionszelle beschreibt. Mit ihr wird der Widerspruch behoben. In Anlehnung an [G13]

damit jedoch nicht modelliert. Die Freiheit, die die Umwelt in der Realität besitzt, sollte im Modell nicht eingeschränkt werden.

2.2. ScenarioTools

ScenarioTools [STHP] ist eine Werkzeugumgebung zur Modellierung und Analyse szenariobasierter Spezifikationen. Dafür bietet es einen grafischen Editor zur Erstellung und Bearbeitung von MSD Spezifikationen. Es besteht die Möglichkeit die erstellten Spezifikationen zu simulieren oder eine Synthese zur Erstellung von Controllern durchzuführen. Die Controller lassen sich als Graphen in das PDF-Dateiformat exportieren und können in dieser Weise als Unterstützung und Orientierungshilfe für die Simulation genutzt werden.

2.2.1. Arbeitsweise

MSD Spezifikationen lassen sich im grafischen Editor als UML Modell beschreiben. Dazu kann die Spezifikation in mehrere Pakete unterteilt werden. Diese Unterteilung gestaltet Modelle übersichtlicher, dient aber auch dazu in einem Modell mehrere Varianten eines Systems beschreiben zu können oder das System in Teilsysteme zu gliedern. In Abbildung 2.7 ist die Struktur eines UML Modells mit mehreren Paketen zu sehen. In dem Paket *Struktur* sind die Klassendiagramme sowie die Objektdiagramme des Modells zusammengefasst. Für das weitere Verfahren ist es ausreichend in den Klassendiagrammen zu beschreiben welche Klassen es gibt sowie in den Objektdiagrammen zu definieren wel-

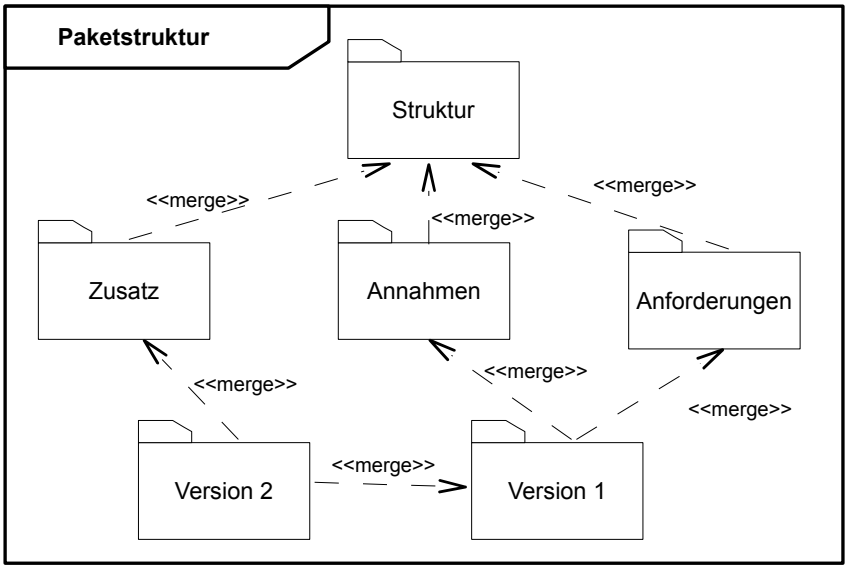
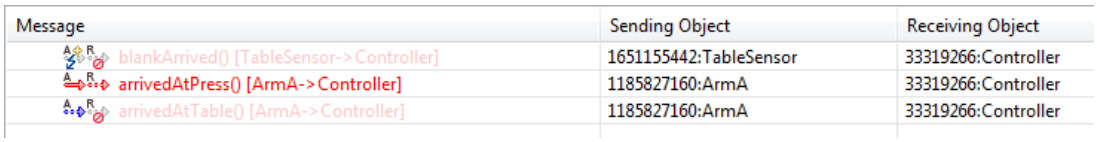


Abbildung 2.7.: Die Paketstruktur einer MSD Spezifikation, die zwei Versionen des Systems enthält



Message	Sending Object	Receiving Object
blankArrived() [TableSensor->Controller]	1651155442:TableSensor	33319266:Controller
arrivedAtPress() [ArmA->Controller]	1185827160:ArmA	33319266:Controller
arrivedAtTable() [ArmA->Controller]	1185827160:ArmA	33319266:Controller

Abbildung 2.8.: Screenshot, der die Nachrichtenauswahl in der Simulation zeigt

che Objekte der Klassen relevant sind und wie diese zu Umwelt und System zugeordnet werden. Im Paket Anforderungen (*Requirements*) befinden sich alle MSDs bzw. MSSs, welche die direkten Anforderungen beschreiben. Das Paket Annahmen (*Assumptions*) beinhaltet alle Annahmen, die über das Verhalten der Umwelt getroffen wurden. Diese Aufgliederung ist lediglich als Beispiel zu verstehen und nicht bindend. Eine Beschreibung von Assumptions und Requirements erfolgte bereits in Kapitel MSD Spezifikation 2.1. Das Paket Version 1 hat die Aufgabe alle relevanten Pakete zusammen zu fassen. Dies geschieht über *merge* Pfeile, die ausdrücken welche Pakete zusammengefasst werden sollen. Das Paket Version 1 enthält nun die erste Version der Spezifikation. Mit dem Paket Version 2 wird eine erweiterte Version beschrieben, die darüber hinaus das Paket Zusatz enthält.

In der *Simulation* können Spezifikationen analysieret werden. Dabei könnten Version 1 oder Version 2 aus Abb. 2.7 als Grundlage dienen. Aus diesem Modell kann das Verhalten des Systems berechnet werden. In jedem Schritt der Simulation wird dem Entwickler eine Menge an Ereignissen zur Auswahl gestellt, die durch die Anforderungen verlangt oder durch die Umwelt ausgelöst werden können. In Abbildung 2.8 ist die Ansicht zu sehen in der der Benutzer die nächste Nachricht auswählen kann. Zu jeder Nachricht

ist das sendende und das empfangende Objekt mit aufgeführt. In den Icons sind die Auswirkungen der Ereignisse auf die Simulation codiert. Dabei gibt es eine generelle Unterteilung in Anforderungen (kurz: R für Requirements) und Annahmen (kurz:A für Assumptions). Der Pfeil repräsentiert den jeweiligen Cut mit seinen Eigenschaften. Ist die Nachricht dabei in mehreren MSDs bzw. MSSs erlaubt, so wird die höherwertige Eigenschaft gewählt. Dabei gilt executed vor monitored und hot vor cold. Ist die Nachricht in einer Anforderung aber in keiner Annahme erlaubt, dann ist der Pfeil auf der Annahmeseite grau. Das Sternchen sagt aus, dass ein neues MSD initialisiert wird, wenn die Nachricht auftritt. Der blaue Blitz steht für eine kalte Verletzung, das rote Verbotsschild steht für eine Sicherheitsverletzung. So kann der Benutzer grob ablesen welche Folgen aus bestimmten Nachrichten resultieren. Soll die Simulation etwas gezielter oder strukturierter ablaufen, kann das Ergebnis der Synthese zur Hand genommen werden. Dies macht insbesondere Sinn, wenn die Spezifikation noch Widersprüche enthält.

Die *Synthese* ist in der Lage aus *realisierbaren MSD Spezifikationen* Steuerregeln für das System abzuleiten. Diese Steuerregeln werden auch *Systemstrategie* genannt. Realisierbar heißt in diesem Zusammenhang, dass die Spezifikationen keinen Widerspruch enthalten. Mit der Systemstrategie kann das System immer wieder Zustände erreichen, in denen es auf alle Nachrichten der Umwelt reagieren kann. Bei MSD Spezifikationen, die unrealisierbar sind, funktioniert die Synthese entsprechend. Allerdings liefert diese ein anderes Ergebnis: Hier wird eine *Gegenstrategie* erzeugt. Eine Gegenstrategie beschreibt wie die Umwelt gegen das System gewinnen kann, indem sie das System in eine Situation bringt, in der das System nur noch die Anforderungen verletzen kann. Dabei kann das System nicht mehr gewährleisten, dass es in Zustände kommt, in denen es auf alle Nachrichten der Umwelt reagieren kann.

Eine Systemstrategie enthält immer alle Entscheidungsmöglichkeiten der Umwelt und die Menge an Systementscheidungen, die für die Erfüllung der Anforderungen nötig sind. Bei der Gegenstrategie ist es genau umgekehrt. Hier hat das System alle Entscheidungsfreiheiten im Rahmen der Anforderungen und die Umwelt ist auf die Nachrichten beschränkt, die zum Verletzen der Anforderungen führen.

Im Folgenden wird erneut das Beispiel aus Kapitel 2.1 betrachtet. Hier konnte bislang das Orakel befragt werden in welcher Situation der Widerspruch entsteht, denn durch einfaches Betrachten der Spezifikation ist dies nicht immer so leicht zu erkennen. Gerade wenn die Spezifikation umfangreicher sind. Genau an dieser Stelle kann die Synthese weiterhelfen. Das Ergebnis der Synthese zu dem Beispiel ohne die 4. Annahme ist in der Abbildung A.2 zu sehen. Der rot umrahmte Knoten markiert immer den Anfangszustand. Rot gefüllte Knoten bezeichnen Zustände, von denen ausgehend das System verlieren kann. Grün gefüllt sind Zustände von denen aus das System andere Zustände erreichen kann, in denen es auf alle Nachrichten der Umwelt reagieren kann. Ein derartiger Zustand ist dann zusätzlich blau umrahmt. Knoten, die rot gefüllt und blau umrahmt sind beschreiben Zustände in denen die Umwelt das System zum Verlieren führen kann. Magenta umrandete Knoten sind Zustände in denen eine Sicherheitsverletzung eingetreten ist. Die Pfeile zwischen den Knoten stellen die Nachrichten dar. Dabei bezeichnen durchgezogene Pfeile Systemnachrichten und gestrichelt gezeichnete Pfeile Umweltnachrichten. An den Pfeilen steht jeweils der Name der Nachricht zusammen mit

dem sendenden und empfangenden Objekt.

Wird die Gegenstrategie vom Startzustand aus verfolgt, ist zu erkennen, dass das System im Zustand 7 die Freiheit besitzt zu entscheiden, ob es zuerst press() oder moveToTable() ausführt. Des Weiteren ist ablesbar in welchem Gesamtzustand sich das System befindet, indem wir alle Nachrichten von Beginn an ausführen. Die Nachricht vor dem Zustand 19 ist die Nachricht, die den Widerspruch verursacht. Mit dieser Gegenstrategie an der Hand kann das System in der Simulation analysiert werden. Dieses Verfahren funktioniert noch sehr gut für kleine System Spezifikationen.

2.2.2. Implementierung

ScenarioTools ist eine Plug-In basierte Erweiterung von Eclipse und im Rahmen der Dissetation von Greenyer [G11] entstanden. Seitdem wurde es stetig weiter entwickelt.

Alle Modelle in ScenarioTools sowie das Programm selbst basieren auf dem *Eclipse Modeling Framework (EMF)*. EMF ist ein Framework zur einheitlichen Modellierung von Modellen und der automatischen Generierung von Code. Dem zu Grunde liegt das *Ecore-Metamodell*, mit dem sich alle Modelle beschreiben lassen. Die MSD Spezifikationen sind als UML Modell mit *Stereotypen* realisiert. Stereotypen wiederum sind ein Formalismus von UML zum Zweck der Erweiterung von UML-Diagrammen. Dem grafischen Editor liegt der *Papyrus Editor* zu Grunde, der für die Erstellung von MSD Spezifikationen erweitert und angepasst wurde.

Bei der Simulation kommt der *Play-Out Algorithmus* von Harel und Marelly [Come, Let's Play] zum Einsatz. Dabei handelt es sich um eine Erweiterung dieses Algorithmus, der unter anderem die in Abschnitt 2.1 beschriebenen Erweiterungen berücksichtigt. Die Simulation läuft dabei auf einem *Laufzeitmodell*, das sich zu den einzelnen Schritten der Simulation die Zustände merken kann. Welches die nächsten gültigen Zustände sind, wird aus den MSD Spezifikationen herausgelesen. Bevor eine Simulation einer MSD Spezifikation gestartet werden kann, muss einmalig ein Wizard ausgeführt werden. Dieser führt eine Transformation von der MSD Spezifikation in ein Objektsystem zusammen mit den zugehörigen Strukturmodellen durch und speichert die Verbindung der beiden Modelle. Dieses Objektsystem wird zur Laufzeit als dynamische Grundlage für die Simulation verwendet, an dem Veränderungen vorgenommen und Zwischenstände gespeichert werden können. Durch die Verlinkung bleibt der Bezug zu der MSD Spezifikation erhalten und kann so als Eingabe für das Play-Out dienen.

Die Synthese arbeitet wie die Simulation ebenfalls auf dem Laufzeitmodell. Zu Beginn wird eine Zustandsraumerkundung durchgeführt. Diese deckt jedoch nicht den gesamten Zustandsraum ab. Die Erkundung endet sobald erkennbar ist, ob die Spezifikation realisierbar bzw. unrealisierbar ist. Dabei wird ein Graph aus verschiedenen Zuständen des Laufzeitmodells auf Grundlage der Regeln der MSD Spezifikation aufgebaut. Wird ein Widerspruch entdeckt, der sich bis zum Startpunkt zurück propagieren lässt, wobei die Umwelt immer wieder die Chance haben muss diesen Pfad einzuschlagen, endet die Zustandsraumerkundung mit dem Ergebnis, dass die Spezifikation unrealisierbar ist.

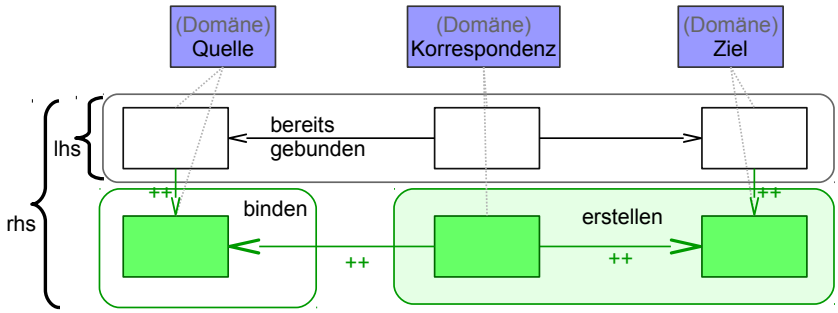


Abbildung 2.9.: Abstrakte TGG Regel, die eine Vorwärtstransformation beschreibt und die drei Teile des Graphen zeigt. In Anlehnung an [GR12]

Werden stattdessen stets Zustände erkundet, in denen das System auf alle Umweltnachrichten reagieren kann und immer wieder in einen solchen Zustand zurückkehren kann, so endet die Zustandsraumerkundung mit dem Ergebnis realisierbar. Im nächsten Schritt wird eine Untermenge von Zuständen und Nachrichten dieser Zustandsraumerkundung gebildet in der nur noch die relevanten Pfade enthalten sind. Bei der Gegenstrategie entfallen alle Nachrichten der Umwelt, die nicht zur Sicherheitsverletzung führen und alle daraus resultierenden nicht erreichbaren Zustände. Entsprechendes gilt für die Systemstrategie. Hier werden alle Nachrichten des Systems entfernt, die zum nicht erfüllen der Anforderungen führen. Ein Beispiel, das das Ergebnis der Zustandsraumerkundung sowie eine Gegenstategie zeigt, ist im Anhang A.1 und A.2 zu sehen.

2.3. Tripel Graph Grammatiken

In diesem Kapitel wird eine Technik zur Modelltransformation vorgestellt. Dabei wird ein Vorgang beschrieben, der sich auf das vorliegende Problem übertragen lässt. ScenarioTools ist nicht in der Lage die Gegenstrategie auszuführen, jedoch ist es in der Lage MSD Spezifikationen auszuführen. Eine Transformation kann immer dann sinnvoll sein, wenn es für das eine Modell effiziente Algorithmen gibt, die ein bestimmtes Problem lösen, für ein anderes Problem jedoch ein anderes Modell bevorzugt wird. Genau diese Vorbedingungen lassen sich in ScenarioTools wiederfinden im Hinblick auf eine Integrierung von Counter-Play-Out.

Die Modelltransformation kann in drei Bereiche eingeteilt werden: Im Allgemeinen wird unterschieden zwischen der *Vorwärtstransformation*, bei der aus einem gegebenen *Quellmodell* ein Modell in einem Zielformat erstellt wird, und der *Rückwärtstransformation*, bei der das *Zielmodell* bereits existiert und Änderungen zurückgeführt werden müssen. Als drittes gibt es noch die Mischung aus beidem, wobei hier Änderungen in Ziel- und Quellmodell möglich sind und die Transformation für die Synchronisation der beiden Modelle sorgt. Im vorliegenden Fall wird die Vorwärtstransformation benutzt.

Tripel Graph Grammatiken (TGGs) [G06] [GR12] sind eine deklarative Modelltransformationssprache. Dabei wird eine Modelltransformation mit einem Satz Regeln beschrieben. Diese Regeln bestehen aus einem dreigeteilten Graphen, aus dem sich auch der Name ableitet. Wie in der Abbildung 2.9 zu sehen, ist auf der linken Seite das Quellmodell, auf der rechten Seite das Zielmodell abgebildet. In der Mitte befindet sich das Korrespondenzmodell. Es speichert welche Elemente des Quellmodells auf welche Elemente des Zielmodells abgebildet werden. Die einzelnen Regeln beschreiben wie Muster aus dem Quellmodell in das Zielmodell abgebildet werden sollen, wobei beide Modelle jeweils zu einer unabhängigen *Domäne* gehören. Eine Domäne beinhaltet dabei das/die Meta-Modell/e mit dem das zugehörige Modell beschrieben ist. Elemente der Domänen werden als Knoten dargestellt und besitzen einen bestimmten Typ, der in der zugehörigen Domäne definiert ist. Die Kanten zwischen den Knoten sind als Referenzen zwischen den Knoten zu verstehen. Blaue Knoten repräsentieren die Domänen. Die weißen Knoten in den TGG Regeln sind *Kontextknoten*. Grüne Knoten stellen *produzierende Knoten* dar und sind mit „++“ gekennzeichnet. Zudem gibt es *produzierende Kanten* und *Kontext Kanten*. Die Markierung ist dabei äquivalent zu der Markierung der Knoten.

Bei TGGs handelt es sich um eine nicht löschende Modelltransformation. In der Abbildung 2.9 sind die Bereiche, die die *left-hand-side* (lhs) und die *right-hand-side* (rhs) überdecken zu sehen. Im Allgemeinen wird der Teil der Regel, der von lhs und rhs überdeckt wird, im Ausgangsgraphen gesucht. Wurde dieser Teil gefunden, dann wird der Teil, der ausschließlich in der rhs enthalten ist, neu erstellt. Alle Elemente, die nur in der lhs enthalten sind, werden gelöscht. Da die lhs in diesen Regeln aber immer eine Teilmenge der rhs ist, wird nie etwas gelöscht. Bei der Interpretation der rhs gibt es bei den TGGs eine Ausnahme. Gehört der zu erstellende Knoten zum Quellmodell, so wird er nicht erzeugt. Stattdessen wird ein ungebundener Knoten im Quellmodell gesucht, der zu diesem Knoten passt und anschließend daran gebunden. Auf diese Weise wird das Quellmodell nicht verändert.

Bei einer Vorwärtstransformation startet die TGG Transformation mit einem *Axiom*. Damit wird ein Startpunkt zwischen dem Quell- und Zielmodell festgelegt. Im Axiom werden die ersten Elemente aus dem Quellgraph gebunden und danach das zugehörige Muster im Zielmodell erstellt. Anschließend wird die Verbindung im Korrespondenzgraph gespeichert.

Im weiteren Verlauf werden darauf aufbauend alle weiteren Regeln so oft angewendet bis keine Regel mehr angewendet werden kann. Beim Anwenden einer Regel wird zuerst nach den Kontextknoten aus der Regel gesucht. Werden sie gefunden, wird nach den ungebundenen Elementen des Quellmodells gesucht, die gebunden werden sollen. Hat diese Suche ebenfalls Erfolg, dann können die Elemente im Ziel- und im Korrespondenzmodell erzeugt werden.

Im Folgenden seien einige erweiterte Sprachkonzepte genannt: *Widerverwendbare Knoten* vereinen die Eigenschaften von Kontext- und Produzierenden Knoten. Sie sind als graue Knoten dargestellt und mit „##“ markiert. Dabei wird der Knoten zuerst als Produzierender Knoten behandelt. Wenn bereits alle Elemente gebunden sind, wird er als Kontextknoten verwendet. Die Bedeutung der *widerverwendbaren Kanten* ist analog zu der Bedeutung der Knoten. Durch den Einsatz von wiederverwendbaren Elementen

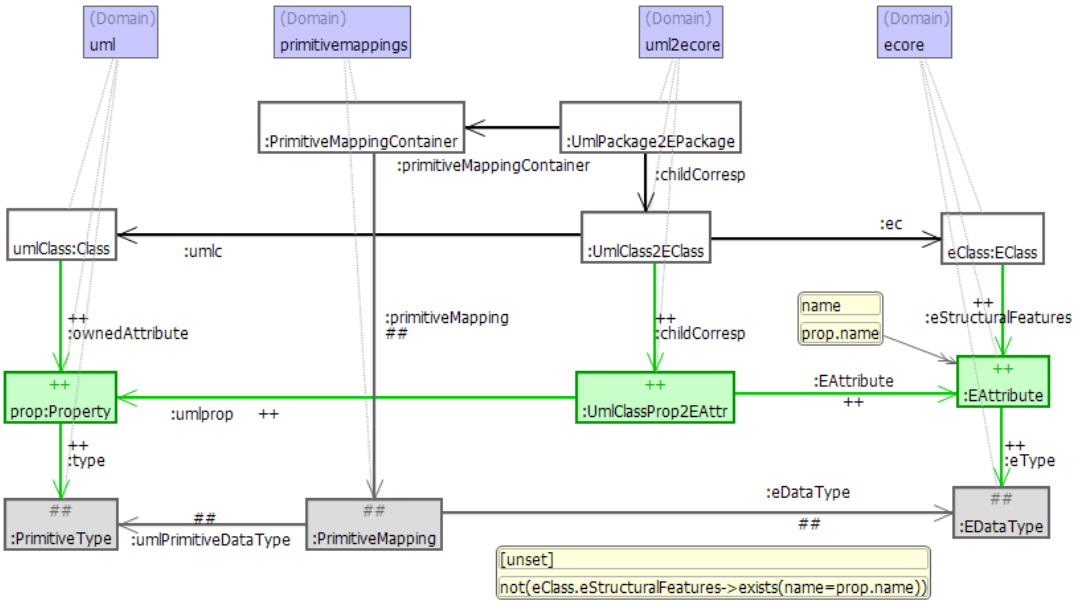


Abbildung 2.10.: Screenshot der TGG Regel *PrimitivePropertyToEAttribute*. Eine Eigenschaft einer UML Klasse wird auf eine Ecore Klasse abgebildet.

steigen die Flexibilität und die Übersichtlichkeit der Regeln.

Zu den erweiterten Funktionen der TGGs zählen zudem noch die Möglichkeit *Object Constraint Language* (OCL) Anfragen zu formulieren und das Binden von Elementen auch für Untertypen zu erlauben.

Im Folgenden wird die Funktionsweise von TGGs an einem Beispiel verdeutlicht. In Abbildung 2.10 ist eine Regel aus dem Regelsatz uml2ecore abgebildet. Mit dieser Transformation kann ein Ecore Klassendiagramm aus einem UML Modell abgeleitet werden. Das UML Modell beinhaltet dabei die MSD Spezifikationen für ein System. Die Transformation wird in ScenarioTools eingesetzt, um Klassen für spätere Objekte zu erstellen, die für die Play-Out Simulation instanziiert werden.

Die in Abbildung 2.10 dargestellte TGG Regel „PrimitivePropertyToEAttribute“ beschreibt die Transformation von einer UML Eigenschaft, die einer Klasse zuzuordnen ist, hin zu einem Attribut, das einer Klasse in Ecore zugeordnet ist. Die blauen Knoten stehen für die verschiedenen Domänen aus denen die Elemente der Transformation stammen. Jeder Knoten muss einer Domäne zugeordnet sein. Diese Zuordnung ist durch die gepunktete Linie beschrieben. In der TGG Regel sind vier Domänen zu erkennen. Die Domäne UML ist die Domäne des Quellmodells. Ecore stellt die Domäne des Zielmodells dar. Die uml2ecore Domäne beschreibt die Korrespondenz zwischen Quell- und Zielmodell. Bleibt noch die Domäne primitivemappings. Sie stellt eine weitere Quelledomäne dar, die einfache (primitive) Datentypen aus UML den zugehörigen Datentypen in Ecore zuordnen kann und umgekehrt.

Soll die Regel während einer Transformation angewendet werden, ist dies nur möglich,

wenn vorher bereits andere Regeln erfolgreich angewendet wurden. Diese Regeln werden an dieser Stelle jedoch nicht genauer erläutert. Für die Folgenden Aspekte ist nun entscheidend, dass die Knoten die weiß erscheinen schon von vorherigen Regeln gebunden wurden sein müssen. Dies sind die sogenannten Kontextknoten. Nach diesen Kontextknoten wird nun als erstes gesucht. In Worten ausgedrückt sucht die Transformation nach einer UML Klasse, die bereits über eine Korrespondenz (UmlClass2EClass) zu einer EClass gebunden wurde. Dabei steht für beide Klassen auch schon fest in welchen Packages sie sich befinden (hier: UMLPackage2EPackage) und diese Package-Korrespondenz weiß wie einfache Datentypen transformiert werden. Nachdem die Kontextknoten gefunden wurden, sucht die Transformation nach einer ungebundenen Eigenschaft (Property) im Quellmodell, die durch einen produzierenden Knoten beschrieben wird. Diese Eigenschaft muss der Klasse bekannt sein, was durch die produzierende Kante ownedAttribute beschrieben wird. Wird solch eine Eigenschaft gefunden, muss diese auch einen Datentyp besitzen. Der Knoten für den Datentyp (PrimitiveType) ist ein wiederverwendbarer Knoten. Allerdings ist der Verweis auf den Datentyp eine produzierende Kante. Der Hintergrund davon ist, dass einfache Datentypen, die in diesen Sprachen verwendet werden, bereits vordefiniert sind und nur einmal abgespeichert werden. Jede Eigenschaft muss ihren Datentyp kennen aber nicht modifizieren. Würde an dieser Stelle einen produzierenden Knoten verwenden werden, hätte dies zur Folge, dass pro Transformation nur maximal so viele Eigenschaften transformiert werden können wie es definierte Datentypen gibt. Dabei darf jeder Datentyp nur einmal vorkommen.

Wurde dieses Muster im Quellmodell gefunden, kann die Transformation ins Zielmodell übergehen. Dazu wird nun die Korrespondenz UmlClassProp2EAttr erstellt. Anschließend wird im Zielmodell ein EAttribut erstellt. Dieses wird mit dem entsprechenden Datentyp verbunden. Zu beachten ist nun noch der gelbe Knoten, der mit dem EAttribut verbunden ist. In dem gelben Kasten steht ein OCL Ausdruck, der sich auf die untergeordnete Eigenschaft des EAttributs *name* bezieht. Der Ausdruck bewirkt, dass das EAttribut in Ecore den gleichen Namen bekommt wie die Property in UML. Jetzt verbleibt noch ein Element, das bisher nicht betrachtet wurde: Der OCL Ausdruck in der rechten oberen Ecke. Hierbei handelt es sich um eine Bedingung, welche die gesamte Regel betrifft. Nur wenn dieser Ausdruck zu wahr evaluiert wird, kann die Regel erfolgreich abgeschlossen werden. Mit dem OCL Ausdruck wird bewirkt, dass kein EAttribut in einer EClass erstellt wird, das den gleichen Namen wie ein bereits vorhandenes besitzt.

3. Anforderungsanalyse

In diesem Kapitel möchte ich die Struktur von ScenarioTools in Bezug auf die Umsetzung der Anforderungen untersuchen und feststellen welche Möglichkeiten sich aus den vorhandenen Strukturen ergeben. Dazu gehe ich kurz auf die Anforderungen ein und analysiere anschließend die relevanten Strukturen von ScenarioTools. Danach führe ich eine Benutzbarkeitsanalyse durch.

3.1. Anforderungsbeschreibung

Ziel dieser Arbeit ist es zwei Aspekte von ScenarioTools zu verbinden. Zum einen kann ScenarioTools mit einem Synthese Algorithmus Controller aus MSD Spezifikationen ableiten. Enthält die Spezifikation einen Widerspruch, so ist der Controller eine Gegenstrategie. Zum anderen können mit Hilfe einer Simulation MSD Spezifikationen ausführbar gemacht werden. Diese beiden Funktionen sollen so miteinander verbunden werden, dass die Simulation den Benutzer mit Hilfe der Gegenstrategie zum Widerspruch führt. Eine Simulation kombiniert mit einer Gegenstrategie wird *Counter-Play-Out (CPO)* genannt. Mit diesem neuen Feature soll dem Benutzer folgende Arbeitsweise mit ScenarioTools ermöglicht werden:

1. Die Spezifikation vom System liegt als Menge von Szenarien vor.
2. Die Szenarien werden vom Benutzer in ScenarioTools in MSDs/MSSs überführt.
3. Die Controllersynthese prüft auf Realisierbarkeit.
4. WENN es einen Widerspruch gibt, DANN wird das Counter-Play-Out ausgeführt SONST Ende.
   - 4.1 Mit Hilfe der Gegenstrategie wird der Benutzer beim Identifizieren des Fehlers unterstützt und bekommt Hilfestellung bei der Behebung des Fehlers.
   - 4.2 Der Benutzer löst den Widerspruch auf.
5. Widerhole den Vorgang ab 3. solange es einen Widerspruch gibt.

Mit dieser Erweiterung soll der Entwicklungsprozess von MSD Spezifikationen verbessert werden. Mit jeder Iteration kann der Benutzer Schwachstellen in seiner Spezifikation erkennen und diese ausbessern. Während des CPO analysiert der Benutzer seine Spezifikation auf eine spielerische Art und Weise. Dabei steuert er die Aktionen des Systems und versucht die Anforderungen zu erfüllen. Die Simulation übernimmt hierbei die Steuerung der Umwelt und versucht ihrerseits die Anforderungen zu verletzen. Allerdings wird der Benutzer gegen die Umwelt verlieren. Genau darin liegt jedoch der

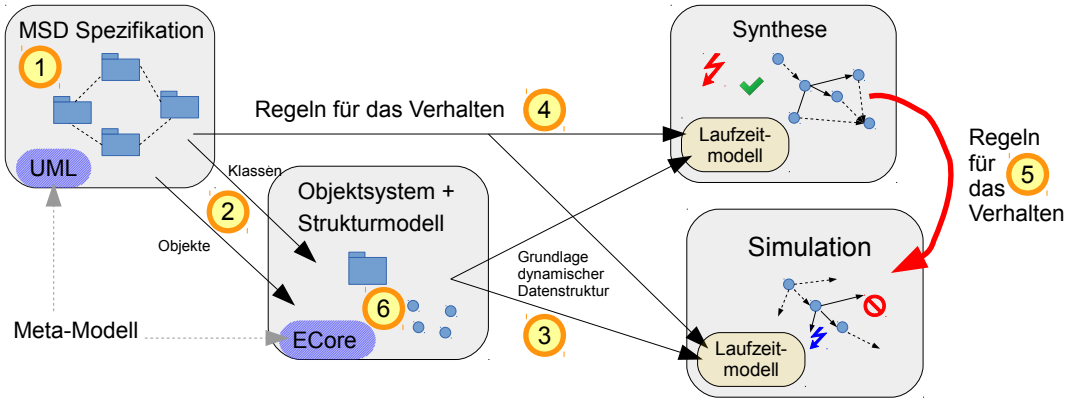


Abbildung 3.1.: Abstraktes Modell, das den Zusammenhang der einzelnen Systemkomponenten von ScenarioTools beschreibt

Erkenntnisgewinn. Wenn der Benutzer im CPO erkennt in welchen Situationen seine Spezifikation Lücken aufweist, kann er diese gezielt beheben.

Folgende weitere Anforderungen an die Umsetzung von CPO sind aus Gesprächen mit meinem Betreuer und der Tatsache, dass mehrere Entwickler an verschiedenen Stellen an ScenarioTools arbeiten, hervorgegangen. Es existieren zahlreiche Erweiterungen, welche die Grundfunktionen von ScenarioTools nutzen und auf die gleiche Basis zurückgreifen. Die nachfolgende Anforderung ist auch als genereller Grundsatz zu betrachten. Sämtliche Daten, die als Eingabe verwendet werden, sollen nur lesend gebraucht werden. Änderungen sollen optional gehalten werden. Es muss davon ausgegangen werden, dass nicht in jedem Anwendungsszenario ein Counter-Play-Out durchgeführt werden soll. Mögliche Wartezeiten, die durch unnötige bzw. ungewollte Transformationen oder ähnliches entstehen, sollen durch vorhergehende Bestätigungsdialoge vermieden werden. Dadurch können gerade bei umfangreicheren Spezifikationen nicht akzeptablen Wartezeiten vermieden werden. Die Aktionen, die für ein CPO notwendig sind, sollen an passender Stelle automatisch ausgeführt und das manuelle Erstellen von Konfigurationsdateien vermieden werden.

3.2. Systemanalyse

Während der Systemanalyse von ScenarioTools hat sich folgender logische Zusammenhang der einzelnen Komponenten ergeben, der in Abbildung 3.1 veranschaulicht wird. Hier wird die Struktur von ScenarioTools in einem sehr abstrakten Modell dargestellt. In (1) ist die MSD Spezifikation zu sehen. Diese beschreibt das Verhalten und die Struktur des Systems in UML. Bevor die Simulation oder eine Synthese durchgeführt werden kann, werden Teile aus dem UML Modell in ein Objektsystem und Strukturmodell überführt (2). Betroffen sind sowohl die Objekte, die simuliert werden als auch die Klassendiagramme, die diese Objekte beschreiben. Dieses Modell basiert nicht mehr auf *UML*

sondern auf *ECore*. In der Abbildung ist dies mit den verschiedenen Meta-Modellen gekennzeichnet. Das Objektsystem wird als dynamische Grundlage für ein Laufzeitmodell verwendet auf dem die Simulation sowie die Synthese aufbauen (3). Werden diese ausgeführt, speichern sie ihren Zustand im Laufzeitmodell und leiten mögliche Folgezustände aus den MSD Spezifikationen ab (4). Das Zwischenergebnis der Synthese ist ein Zustandsgraph aller erkundeten Zustände. Der Controller bildet eine Untermenge davon und beinhaltet eine Kette von Ereignissen, die die Strategie beschreibt. Für die Strategie gibt es derzeit keinen Interpreter, der sie ausführen kann. Die Simulation kann nicht auf die Strategie zugreifen(5), da sie sich in einem anderen Modell befindet. Von dort aus kann sie kein Verhalten ableiten. Dieses Problem soll gelöst werden. Der rote Pfeil kennzeichnet die zu schließende Lücke.

Die Dateien aus der Speicherebene lassen sich ebenfalls der Abb. 3.1 zuordnen. Diese sind für die Transformation interessant, da sie mögliche Domänen darstellen. Im Folgenden werden die während eines Vorgangs erzeugten Dateien aufgezeigt:

***.uml** Die UML Datei (1) in 3.1 enthält die MSD Spezifikationen. Diese Datei wird vom Benutzer erstellt und bearbeitet.

Folgende sechs Dateien können von einem Wizard erzeugt werden:

***.interpreterconfiguration** Mit der Interpreter Konfiguration (2) in 3.1 kann die uml2ecore-Transformation durchgeführt werden. Dabei werden aus der MSD Spezifikation Klassendiagramme abgeleitet. Das Ergebnis wird in der *.ecore Datei gespeichert (siehe *.ecore).

***.ecore** Diese Ecore Datei (6) in 3.1 ist das Ergebnis der uml2ecore-Transformation und enthält die Struktur der Objekte der MSD Spezifikation. Sie bildet die Grundlage für das Laufzeitmodell.

***.corr.xmi** Die *.corr.xmi Datei (2) in 3.1 bildet den Korrespondenzgraphen der TGG-Transformation uml2ecore ab.

***.xmi** In dieser xmi Datei (6) in 3.1 sind die entsprechenden Objekte der MSD Spezifikation als ECore Objekte gespeichert. Sie werden aus der MSD Spezifikation abgeleitet und geben vor welche Objekte im Laufzeitmodell instanziiert werden.

***.roles2eobjects** In der roles2eobjects Datei (2) in 3.1 ist die Abbildung von UML Objekten auf ECore Objekte gespeichert.

***.scenariorunconfiguration** Die scenarioRunConfiguration Datei fast alle Dateien für die Simulation und die Synthese zusammen. (corr.xmi, roles2eobjects, untermenge von xmi, und weitere Einstellungen)

***.msdruntime** Die Datei msdruntime enthält einen Stategraph. Hier wird das Ergebnis der Zustandsraumerkundung gespeichert. Im vorherigen oft als Zwischenergebnis der Synthese bezeichnet.

***Controller.msdruntime** Die Controller.msdruntime Datei enthält die Strategie. Die Datei beinhaltet Verweise auf den StateGraph der Zustandsraumerkundung. Daten werden dadurch nur einmal gespeichert.

Daraus ergeben sich zwei mögliche Ansätze eine Strategie in die Simulation mit einzu-

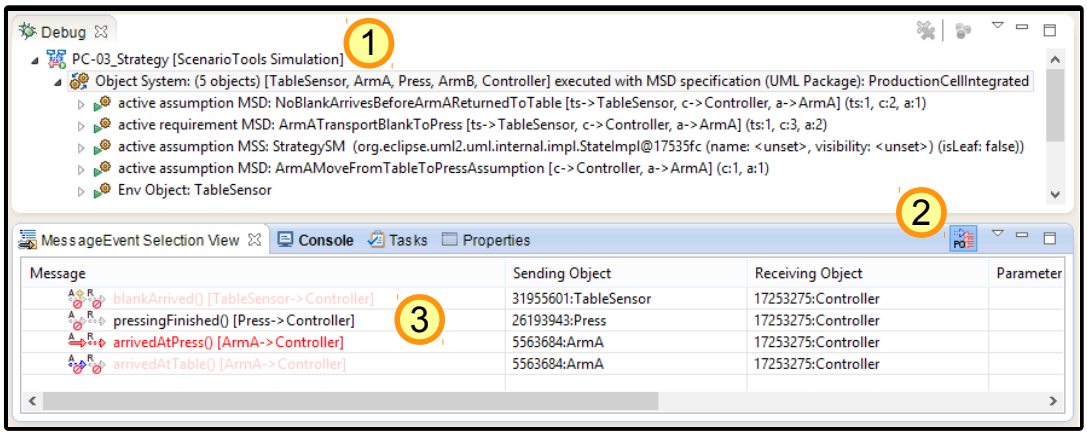


Abbildung 3.2.: Simulationsansicht von ScenarioTools in der Problembereiche kenntlich gemacht sind

beziehen:

1. Implementation eines Interpreter für die Strategie (Strategie-Interpreter)
2. Durchführung einer Transformation der Strategie zu UML

Zu 1. Strategie-Interpreter Ein Interpreter für die Strategie würde parallel zur Simulation ausgeführt werden. Ziel des Interpreters ist es, die Auswahl der möglichen Nachrichten entsprechend zu filtern oder eine Priorisierung vorzunehmen. Dabei würde ein Abgleich der Nachrichten aus der Strategie mit den erlaubten Nachrichten der Simulation stattfinden.

Zu 2. Transformation Die Strategie könnte zurück zu UML transformiert werden. Da sie sich in einer zustandsähnlichen Struktur befindet, könnte sie in ein MSS überführt werden. Das MSS würde dann mit der MSD Spezifikation zusammen ausgeführt werden. Die Interpretation übernähme dabei die Simulation.

3.3. Benutzbarkeitsanalyse

In diesem Abschnitt möchte ich die Benutzeroberfläche in Bezug auf die Eignung zur Simulation analysieren.

Bei der Analyse der Benutzeroberfläche von ScenarioTools hat sich herausgestellt, dass für die Ausführung des Counter-Play-Out keine Anpassungen nötig sind. Allerdings hat sich die Benutzeroberfläche bei der ersten prototypischen Ausführung eines Counter-Play-Out als nicht sehr intuitiv erwiesen. In Abbildung 3.2 ist die Benutzeroberfläche von ScenarioTools dargestellt. Zu sehen ist die *ScenarioTools MSD Simulation Perspective*. Sie definiert die Oberfläche und besteht aus mehreren Views. Im oberen Teil der Grafik ist der *Debug View* zusehen. Hier sind das Objektsystem und die aktiven MSDs bzw. MSSs der Spezifikation dargestellt. Im unteren Teil ist der *MessageEventSelection-*

View zu sehen, in dem die Nachrichten angezeigt werden. Dabei gibt es zwei Modi: Im Play-Out Modus werden nur die Nachrichten angezeigt, die erlaubt sind. Im General Modus werden die erlaubten Nachrichten und zusätzlich alle Umweltnachrichten angezeigt. Folgende Probleme haben sich bei der Analyse herausgestellt:

(1) Während die Simulation läuft ist nicht erkennbar, um welche Art von Simulation es sich handelt.

(2) Die Simulation sollte im Play-Out Modus ausgeführt werden, weil im General Modus immer alle Umweltnachrichten zusätzlich zur Auswahl gestellt werden. Dadurch wird die Wahrscheinlichkeit gesteigert, dass der Benutzer ungewollt die falsche Nachricht auswählt und das Resultat entsprechend falsch interpretiert.

(3) Bei der Auswahl der Nachricht ist das Risiko sehr hoch, dass der Benutzer die falsche Nachricht auswählt. Tritt dieser Fall ein, muss die Simulation neu gestartet werden, da es keine Funktion zum Rückgängig machen gibt.

(3.1) Die Aussage der Icons wird durch die Strategie verändert. Die Icons beschreiben nun weniger die MSD Spezifikation, sondern vielmehr die von der Strategie vorgesehenen Nachrichten.

(3.2) Des Weiteren muss dem Entwickler klar sein welches Ziel er bei der Simulation verfolgt, sprich welche Art von Verletzung er erzwingen oder vermeiden möchte. Dieser Punkt hängt eng mit (1) zusammen.

4. Entwurf

In der Analyse haben sich zwei mögliche Lösungsansätze herausgestellt. Zum einen kann eine Strategie in die Simulation Integriert werden, indem sie von einem Strategie-Interpreter eingelesen wird und dieser dann die Simulation manipuliert. Zum anderen kann die Strategie in einer Transformation in ein MSS überführt werden und anschließend gleichberechtigt mit der MSD Spezifikation ausgeführt werden.

Im Folgenden habe ich mich dafür entschieden die Umsetzung mit Hilfe einer TGG Transformation durchzuführen. Die Entscheidung ist auf TGGs gefallen, da diese einige Vorteile mit sich bringen. ScenarioTools enthält bereits ein Plug-In mit dem TGG Transformationen durchgeführt werden können: Den TGG Interpreter [G06]. Ein weiterer Grund für diese Entscheidung ist die Tatsache, dass die Erweiterung auf diese Weise als weiteres Plug-In realisiert werden kann.

Da die Strategie gleichberechtigt mit der MSD Spezifikation von der Simulation eingelesen wird, muss keine Schnittstelle zwischen Simulation und einem Strategie-Interpreter geschaffen werden. Zudem muss der Kern der Simulation dabei nicht erweitert werden. Bei der Integration der Strategie in die Simulation besteht die Schwierigkeit darin, die Nachrichten zwischen den beiden Modellen (UML und Ecore) richtig zuzuordnen. Diese Hürde muss von beiden Varianten überwunden werden.

Die Implementierung eines Strategie-Interpreters hätte den Vorteil flexibler zu sein. Ein Interpreter wäre in der Simulation nicht auf die Möglichkeiten einer MSS beschränkt. Dem gegenüber steht jedoch ein höherer Implementierungs- und Wartungsaufwand des Strategie-Interpreters. Zudem sind diese Möglichkeiten eines MSS zum jetzigen Zeitpunkt ausreichend.

Es wäre möglich im Zuge der Erweiterbarkeit der Strategie später zusätzliche Informationen anzuhängen. Dies kann mit *EAnnotation* umgesetzt werden. EAnnotation sind Notizen, die einzelnen Elementen angehängt werden können, um beispielsweise eine differenzierte Betrachtung durch die Simulation herbeizuführen. Diese könnten dann in der Simulation mittels einer kleinen Erweiterung ausgelesen werden.

Im Folgenden ist der Entwurf zu sehen. Zuerst werde ich betrachten, welche allgemeinen Anforderungen ein MSS erfüllen muss, damit es die Aufgabe der Strategie erfüllen kann. Anschließend werde ich die Integration der Transformation in den Ablauf von ScenarioTools erläutern und schließlich den wichtigsten Teil der Transformation vorstellen.

4.1. Strategie als MSS

In diesem Abschnitt soll eine Strategie auf ein MSS abgebildet werden. Das MSS, das die Strategie beinhaltet, wird der MSD Spezifikation zugeführt und von dort an gleichberechtigt behandelt. Ziel dieses Abschnittes ist es ein MSS zu modellieren, das die Aufgaben einer Gegenstrategie sowie einer Systemstrategie erfüllt. Im Folgenden werde ich Strategie als Synonym für beide Strategietypen verwenden. Wenn Unterschiede bestehen sind diese genannt.

Eine Strategie ist eine zustandsbasierte Beschreibung von erlaubten Nachrichten in Form eines Graphen. Die Übergänge sind die Nachrichten. Von einem Zustand aus betrachtet können alle ausgehenden Nachrichten potentiell als nächstes geschehen. Gleichzeitig führen sie zu einem Folgezustand. Ein MSS besitzt eine sehr ähnliche Grundstruktur.

Bei einer Gegenstrategie ist das Verhalten der Umwelt in einem bestimmten Zustand immer eindeutig beschrieben. Die Handlungsfreiheit des Systems wird nicht eingeschränkt. Es kann allerdings passieren, dass das System einzelne Zustände nicht mehr erreichen kann, weil die Umwelt es nicht zulässt bzw. unterstützt. Für den Fall einer Systemstrategie gilt das Gleiche mit vertauschten Rollen.

Wenn die Strategie als MSS zusammen mit der MSD Spezifikation ausgeführt wird, steht sie gleichberechtigt da. Bei der Ausführung soll das Verhalten durch die Strategie eingeschränkt werden. Es darf zu keiner Erweiterung des Verhaltens kommen. Die Strategie wird beim Auftreten der ersten Nachricht aktiviert und beeinflusst von dort an das Handeln. Da die Strategie aus der MSD Spezifikation abgeleitet wurde, enthält sie nur Verhaltensabläufe, die durch die MSD Spezifikation erlaubt sind.

Allerdings kann dieser Punkt nur gewährleistet werden, wenn die Strategie mit der ersten Nachricht aktiviert wird und zwischenzeitig nicht unterbrochen wird. Wird die Strategie unterbrochen und anschließen nicht im Startzustand der Simulation wieder aktiviert, kann es zu Seiteneffekten kommen. Zu diesem Zeitpunkt kann nicht mehr gewährleistet werden, dass keine zusätzlichen Handlungsmöglichkeiten durch die Strategie geschaffen werden oder die Handlungsanweisungen der Strategie den Anweisungen der MSD Spezifikation widersprechen. Damit gewährleistet werden kann, dass die Strategie nicht vorzeitig beendet wird, werden alle Zustände der Strategie auf *hot* und *executet* gesetzt. Daraus folgt, dass ein Verletzen der Strategie eine Sicherheitsverletzung mit anschließendem Simulationsende bewirken würde.

Häufig tritt eine Verletzung der Spezifikation auf, weil ein Ereignis der Umwelt nicht auftritt. Dann liegt der Fehler, dass das Ereignis nicht auftritt, allerdings nicht an der Tatsache, dass es explizit verboten wurde, sondern einfach daran, dass es nicht explizit gefordert wurde. Eine solche Nachricht ist demzufolge auch kein Bestandteil der eigentlichen Gegenstrategie, da sie durch Auslassen dieser Nachricht zum Widerspruch gelangt. Bei der Simulation wird immer die Gesamtmenge aller erlaubten Nachrichten gebildet, da die Strategie gleichberechtigt interpretiert wird. So kann es zu einem bestimmten Zeitpunkt der Simulation dazu kommen, dass die Strategie ein bestimmtes Verhalten fordert, die MSD Spezifikation jedoch ein erweitertes Verhalten anbietet. Aus diesem Grund muss die Strategie immer alle potentiell in der Simulation auftretenden Nach-

richten berücksichtigen. Andernfalls kann die Simulation in einen Zustand gelangen, den die Strategie nicht abbilden kann. Diese Eigenschaft kann ein MSS nur erfüllen, wenn es auch alle Nachrichten enthält. Zu diesem Zweck kann ein nicht erreichbarer Zustand eingefügt werden. Dieser nimmt alle Nachrichten oder nur die nicht in der Strategie enthalten Nachrichten als Ausgangs- und Folgezustand auf. Auf diese Weise reagiert das MSS auf jede Nachricht. Ist die Nachricht in der Strategie erlaubt, so erlaubt es auch das MSS. Wenn die Nachricht in der Strategie nicht erlaubt ist, führt ihre Ausführung zur Sicherheitsverletzung.
Systemstrategien werden als Anforderungen modelliert. Die Gegenstrategie beschreibt das Verhalten der Umwelt. Aus diesem Grund wird sie als Umweltannahme modelliert.

4.2. Integration

Die Transformation soll direkt im Anschluss an die Synthese erfolgen. Damit die neue Funktion vorerst optional bleibt, soll der Benutzer die Transformation bestätigen. In Abbildung 4.1 ist beschreiben wie die Transformation in den bestehenden Prozess integriert werden soll.

Nachdem der Benutzer auf einer ScenarioRunConfiguration Datei das Event On-The-Fly Synthese ausgelöst hat, startet der Synthesejob (1) mit der Datei als Eingabeparameter. Der Synthesejob erzeugt zwei Dateien. Zum einen wird das Zwischenergebnis der Synthese, der (MSDRuntime-)StateGraph, der die erkundeten Zustände beinhaltet, gespeichert. Zum anderen wird der synthetisierte Controller ebenfalls als (Runtime-) StateGraph gespeichert. Ist der Synthesejob beendet, wird der Benutzer über das Ergebnis informiert. Anschließend kann der Benutzer entscheiden, ob die Transformation ausgeführt werden soll. Entscheidet er sich dagegen, endet der Gesamtprozess an dieser Stelle. Entscheidet er sich dafür, wird die strategy2mss Transformation (2) gestartet. Dafür benötigt der Prozess die ScenarioRunConfiguration Datei sowie den synthetisierten Controller als Eingabeparameter. Das Ergebnis der Transformation wird ebenfalls in Dateien gespeichert. So produziert die Transformation den Controller als MSS sowie eine Korrespondenzdatei, welche die Abbildung der Strategie als StateGraph zum MSS speichert. Außerdem wird eine weitere ScenarioRunConfiguration erzeugt, mit der die Strategie zusammen mit der MSD Spezifikation simuliert werden kann. Um die Strategie mit in die Simulation integrieren zu können, muss sie der Simulation übergeben werden. Dies könnte auf mehrere Arten geschehen:

1. Erstellung eines neuen UML Modells, das die Strategie enthält und die MSD Spezifikation integriert. Dies könnte beispielsweise über eine Paketvereinigung geschehen.
2. Die Strategie wird in vorhandene MSD Spezifikation integriert.
3. Erweiterung der Szenariorunkonfiguration um einen Verweis auf die Strategie sowie Erweiterung der Simulation dahingehend, dass das Strategiepaket zusätzlich initialisiert wird.

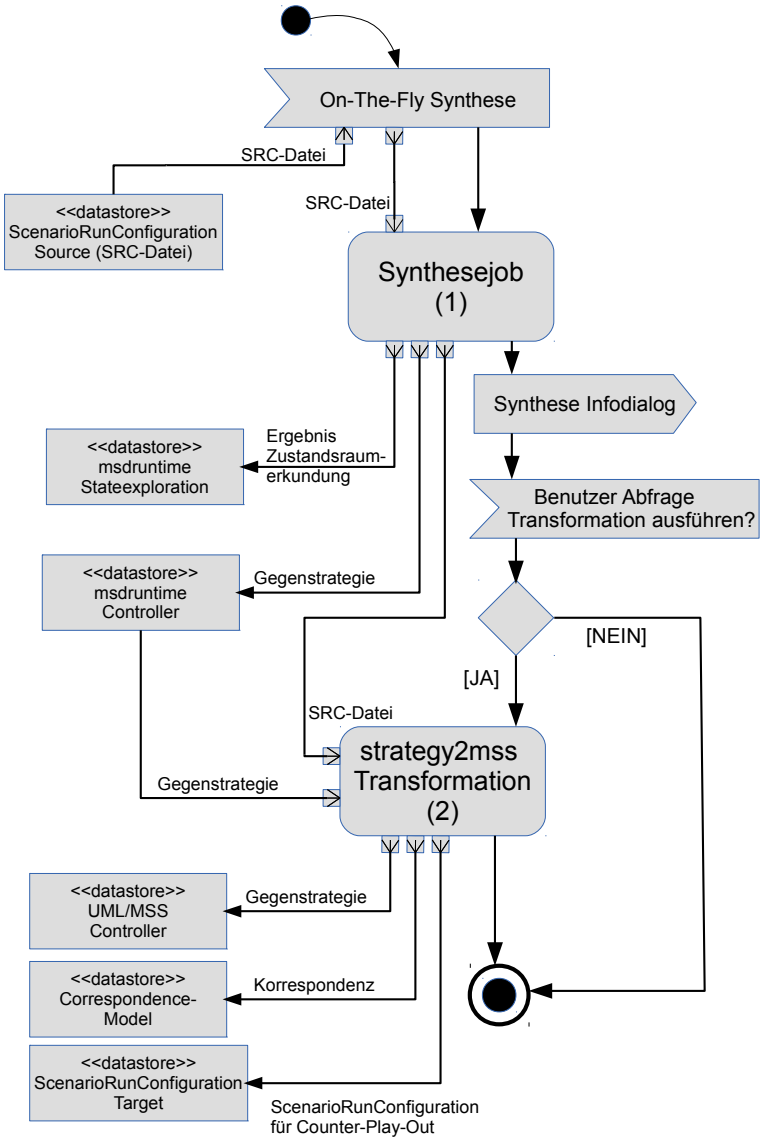


Abbildung 4.1.: Das Aktivitätsdiagramm beschreibt den Ablauf, in den die Transformation integriert wurde

Ich habe mich zur Integration der Strategie für Variante 3 entscheiden. Variante 1 scheidet aus, da die Simulation eine Paketvereinigung über mehrere Dateien nicht unterstützt. Die zweite Variante steht im Widerspruch zu der Anforderung, dass Eingabedaten nicht verändert werden sollen.

4.3. Strategy2mss Transformation

In diesem Abschnitt beschreibe ich die technische Umsetzung der Transformation. Ziel hierbei ist es die Strategie nach dem Vorbild des vorherigen Abschnittes 4.1 abzubilden. Für die Transformation benutze ich den TGG Interpreter wie er im GrundlagenKapitel 2.3 beschrieben wurde.

Die Transformation der Strategie wird mit dem TGG Interpreter vorgenommen. Dabei soll eine Vorwärtstransformation durchgeführt werden, wobei nach Abschluss der Transformation die Strategie als MSS abgebildet ist. Die Wahl fällt auf eine Vorwärtstransformation, weil die Strategie als Quellmodell gegeben ist und hierbei das Zielmodell erzeugt wird.

4.3.1. Bestimmung der Domänen

Beginnen wir nun damit die Domänen des Quell- und Zielmodells festzusetzen. Als Domäne verwende ich jeweils das Paket, in dem die Datenstruktur, mit der die jeweilige Eigenschaft beschreiben wird, enthalten ist. Die Strategie wird durch einen StateGraph beschrieben, der sich in dem Pakete Stategraph befindet. Die Quelldomäne erhält die Dateiendung der Datei, in der sich die Strategie befindet, als Name *msdruntime*. Das Zielmodell ist die MSD Spezifikation, die in UML mit dem Stereotypen *modal* beschrieben ist. Die Zieldomäne erhält deshalb den Namen *uml*. Ein MSS ist dabei als Endlicher Automat mit einem Stereotyp modelliert. Die Korrespondenzdomäne trägt den Namen *strategy2mss*.

Betrachten wir einen einfachen Zustand, mit dem Wissen, dass dieser kaum Informationen enthält, so können wir mit dieser Erkenntnis Zustände aus der Strategie in Zustände eines MSS überführen.
Schauen wir uns als nächstes die Transformation der Nachrichten an. Eine Nachricht in MSD Spezifikationen hat einen Sender und einen Empfänger. Zudem besitzt sie einen Namen. Dieser Name bezeichnet eine Funktion, die zum empfangenden Objekt gehört. Ferner ist die Nachricht Teil eines Objektsystems, das zur Laufzeit betrachtet wird. Es gibt also keine statische Zuweisung eines Nachrichtentyps zu einem Objekt. Die Nachrichten der Strategie haben diese Eigenschaften ebenfalls, jedoch beziehen sie sich auf ein anderes Objektsystem. Für die Ausführung des CPO ist es allerdings wichtig, dass die Zuordnung der Nachrichten korrekt abgebildet wird. Die Strategie soll in das Objektsystem der MSD Spezifikation übernommen werden und dabei ihre Bedeutung nicht verlieren. Dazu kann ausgenutzt werden, dass das Laufzeitmodell, auf dem die Synthese arbeitet, von der MSD Spezifikation abgeleitet wurde und über die Rückverfolgung dieses Weges eine Zuordnung der Objektsysteme getroffen werden kann.

Betrachten wir die Abbildung 4.2. Das Schaubild soll den Zusammenhang der MSD Spezifikation mit der Synthese und der anschließenden Transformation verdeutlichen. Zu sehen ist die MSD Spezifikation (1). Sie bildet die Grundlage. Mit der UML2Ecore-Transformation werden Klassendiagramme aus UML in Ecore abgebildet. Bei dieser Transformation handelt es sich ebenfalls um eine TGG-Transformation. Dabei reprä-

sentiert der Datenbehälter (2) corr.xmi den Korrespondenzgraphen. Hierin sind alle interessanten Abbildungen zwischen UML und Ecore, wie zum Beispiel die Zuordnung von Klassen oder Operationen, gespeichert. Roles2EObjects (3) beschreibt eine Abbildungsfunktion, die Objektzuordnungen zwischen UML und Ecore durchführen kann. Diese Objekte sind ebenfalls aus der MSD Spezifikation abgeleitet. Während der Synthese werden die Objekte instanziiert und bilden das Laufzeitmodell (4). Hier werden die verschiedenen Zustände und Nachrichten, die während der Synthese erkundet werden, in einer Datenstruktur (5) festgehalten. Die Strategie (6) wird aus einer Untermenge dieser Zustände gebildet. Soll an dieser Stelle der Kreis geschlossen (7) und die Strategie mit der MSD Spezifikation verbunden werden, kann der Weg zurückgegangen und die entsprechenden Zuordnungen über den Korrespondenzgraphen der UML2Ecore Transformation und die Roles2EObjects Abbildung getroffen werden. Auf diese Weise kann sichergestellt werden, dass die Objekte der Strategie auf die richtigen Objekte in der Spezifikation abgebildet werden.

Damit unsere Domäne des Quellmodells diese Zuordnung abdecken kann, müssen wir sie um die nötigen Informationen erweitern. Dazu muss zusätzlich zur Strategie das Laufzeitmodell mit in Betracht gezogen werden. Das Laufzeitmodell wird durch die Meta-Modelle msd.runtime und runtime beschrieben. Nachrichten, die hier auftreten, werden durch das Meta-Modell events beschrieben. Die Objekte des Objektsystems werden als EObjects generalisiert. Dafür wird auf das Ecore Meta-Modell zurückgegriffen. Damit ist der Bereich der Synthese und der Simulation abgedeckt.

Die Zuordnung zwischen UML und dem Laufzeitmodell habe ich als eigene Domäne modelliert. Der Grund dafür ist, dass diese Elemente nicht direkt zum Quellmodell gehören. Zudem sind die Elemente im Dateisystem in unterschiedlichen Dateien gespeichert. Dieser Umstand wird durch eine zusätzliche Domäne verdeutlicht. Die Zuordnung zwischen UML und Laufzeitmodell werden durch die Pakete UML2Ecore und roles2Eobjects beschrieben. Zusätzlich wird die ScenarioRunConfiguration als Eingabe für die Transformation benötigt, damit aus ihr die Konfiguration für ein Counter-Play-Out oder ein Strategy-Play-Out abgeleitet werden kann. Da die ScenarioRunConfiguration die beiden anderen Pakete indirekt enthält, wähle ich *ScenarioRunConfigurationSource* als Domänenbezeichner. Die Zieldomäne, welche die abgeleitete Konfiguration enthält, trägt den Namen *ScenarioRunConfigurationTarget*. Sie muss die Pakete UML2Ecore und roles2Eobjects nicht kennen, da sie sie nur verlinkt.
In Tabelle 4.1 ist die Zuordnung der Domänen zusammengefasst.

4.3.2. Ergebnis der Transformation

Das Ergebnis der Transformation wird in mehreren Dateien abgespeichert. Dabei werden die Dateinamen und –pfade nach folgendem Muster erstellt: Alle erstellten Dateien werden im gleichen Ordner gespeichert. Es handelt sich hierbei um den Ordner, der die ScenarioRunConfiguration-Datei beinhaltet, auf der die Synthese ausgeführt wurde. Die Dateinamen bestehen aus dem Dateinamen der ScenarioRunConfiguration-Datei oh-

Tabelle 4.1.: Zuordnung der Domänen

Domäme	Name	Meta-Modell
Quelldomäne	msdruntime	runtime, runtime, stategraph, events, ecore
Quelldomäne	scenarioRun-ConfigurationSource	scenariorunconfiguration, roles2eobjects, uml2ecore
Korrespondenzdomäne	strategy2mss	strategy2mss
Zieldomäne	scenarioRun-ConfigurationTarget	scenariorunconfiguration
Zieldomäne	uml	uml, Modal

ne Endung. Diese werden um das Wort *Controller* sowie der passenden Dateiendung ergänzt.

***Controller.uml** Die Datei *Controller.uml enthält ein UML Modell, das aus einem Paket besteht. In diesem Paket befindet sich die Strategie als MSS.

***Controller.corr.xmi** In der *Controller.corr.xmi-Datei befindet sich der Korrespondenzgraph der Transformation. Hier sind die Informationen darüber gespeichert wie die Strategie als MSS mit der Strategie als StateGraph, der in der msdruntime Datei gespeichert ist, zusammenhängt.

***Controller.scenariorunconfiguration** Die *Controller.scenariorunconfiguration Datei beinhaltet die Konfiguration für die Play-Out Simulation. Zusätzlich zu den Informationen, die erforderlich sind, um eine MSD Spezifikation auszuführen, enthält sie den Verweis auf die Strategie.

4.3.3. Umsetzung der TGG Regeln

Im Folgenden stelle ich stellvertretend für die strategy2mss Transformation zwei Regeln der Transformation vor.

In Abbildung 4.3 ist das Axiom der TGGs zu sehen. Dies ist die TGG Regel, die immer als Erstes ausgeführt wird. Aus diesem Grund enthält sie weder Kontext- noch wiederverwendbare Knoten. Deutlich erkennbar sind die Domänen der Modelle. Mit dem Axiom werden die Wurzelelemente der einzelnen Domänen erzeugt bzw. gebunden. Ganz links ist das Quellmodell, die Strategie, zu sehen. Daneben steht die ScenarioRun-Configuration zusammen mit ihrer Domäne. Die Konfiguration kennt die UML Objekt zu EObject Abbildung (RoleToEObjectMappingContainer). Können diese drei Elemente im Axiom gebunden werden, kann die TGG Transformation angewendet werden. Dazu wird als Erstes die benötigte Grundstruktur im UML Modell angelegt. Als oberstes Element hat ein UML Modell einen Container vom Typ Modell. Darin befindet sich ein Paket, in dem sich ein Element von Typ Kollaboration befindet. Anschließend wird die Konfiguration im Zielmodell angelegt, die nun die gleiche UML Objekt-zu-EObject-Abbildung wie die Konfiguration im Quellmodell kennt. Zusätzlich besitzt die neue Konfiguration einen

Verweis auf das Paket, in dem sich später der Endliche Automat(, bzw. nach Anwendung des Stereotypen das MSS) mit der Strategie befindet wird. In der Mitte der Regel ist des Weiteren der Korrespondenzgraph der Regel zu sehen. Er speichert sämtliche Referenzen zu allen Elementen, die im Axiom gebunden wurden. Das Speichern dieser Informationen erleichtert in den folgenden Regeln die Verknüpfung von Elementen. Da das Axiom nur einmal angewendet wird, sind die Informationen in den weiteren Regeln eindeutig zuzuordnen.

Die zweite TGG Regel in Abbildung 4.4 beschreibt wie Übergänge aus der Strategie in das MSS abgebildet werden. Dabei ist an jeden Übergang eine Nachricht gekoppelt. In dieser Regel geschieht die Hauptarbeit. Mit ihr wird der Kreis geschlossen und sichergestellt, dass die Objektsysteme korrekt aufeinander abgebildet werden. Bevor die Regel angewendet werden kann, müssen vorher alle Regeln, die die Kontext-Knoten der Regel binden und erstellen, erfolgreich angewendet worden sein. Wenn die Regel angewendet wird, wird nach Zuständen, die bereits abgebildet wurden, gesucht. Wird in der Strategie zwischen zwei Zuständen ein Übergang, zu dem eine Nachricht, das SynchronousMessageEvent, gehört, gebunden, so kann im MSS ebenfalls ein Übergang zwischen den entsprechenden Knoten der Abbildung erstellt werden. Dieser Übergang wird in der Region gespeichert. Anschließend muss die Nachricht in das andere Objektsystem abgebildet werden. Dabei werden im Quellmodell die gebundenen Elemente der Nachricht betrachtet. Die Nachricht kennt sein sendendes und sein empfangendes Objekt. Diese werden mit der UML-Objekt-zu-EObjekt-Abbildung aus der ScenarioRunConfigurationSource Domäne auf das richtige Objekt in UML abgebildet und hier mit der MessageEventTransition verbunden. Dieses Element ist Teil des Stereotypen und erweitert die Transition.

Im Gegensatz zu den Objekten, bei denen der direkte Verweis gespeichert wird, wird der Funktionsname im MSS nur als String angegeben. Aus der Kombination Funktionsname und empfangendes Objekt kann die Operation bei der Interpretation eindeutig bestimmt werden. An dieser Stelle wäre es auch denkbar die Operation als direkten Verweis zu speichern. Damit ließe sich eine Performancesteigerung erzielen, da der Aufwendige String Vergleich entfallen würde. Welcher aus der jetzigen Implementierung resultiert. Allerdings müsste für die Speicherung des Verweises zuerst der Stereotyp angepasst werden. Dies wiederum würde den Rahmen dieser Arbeit überreizen. Als zweite Möglichkeit bietet sich die Verwendung von EAnnotation an. Dabei müsste die Interpretation leicht angepasst werden, sodass die EAnnotation berücksichtigt werden. Bei dieser Lösung könnte der Operationsname als String jedoch nicht entfallen da er sonst nicht im grafischen Editor für die Bearbeitung der MSD Spezifikation angezeigt und geändert werden kann. Eine Implementierung bei der beide Möglichkeiten berücksichtigt werden würde keinen Performancegewinn bringen. Da es nur eine Strategie gibt, aber viele Diagramme der MSD Spezifikation, würde bei der Großzahl der Fälle vergebens auf eine EAnnotation geprüft werden.

4.4. Benutzeroberfläche

Bei dem Entwurf der Oberfläche steht die Verbesserung der Usability im Vordergrund. Dabei möchte ich die Probleme aufgreifen, die ich bereits in Abschnitt 3.3 erläutert habe.
Der in Abbildung 3.2 dargestellte Punkt (1) soll um die Informationen, die beschreiben um welche Simulationsart es sich handelt, angereichert werden. Auf diese Weise soll die Übersicht verbessert werden. Der Titel entspricht folgendem Muster:

```
[Name des Projekts] [ScenarioTools
                        {Simulation | Strategy-Play-Out | Counter-Play-Out}]
```

Zu Punkt (2) und (3): An dieser Stelle habe ich mich dafür entschieden alle Nachrichten, die nicht von der Strategie erlaubt sind, zu deaktivieren. Die Nachrichten werden noch angezeigt, können jedoch nicht ausgewählt werden. Des Weiteren soll der aktuelle Zustand der Strategie nicht für die Zusammensetzung der Icons berücksichtigt werden. Somit können durch die Icons wieder Rückschlüsse auf die Spezifikation gezogen werden. Dies soll erreicht werden indem EAnnotation an die Nachrichten der Strategie angehängt werden. Wird eine Strategie ausgespielt, werden alle Nachrichten auf EAnnotation mit einem bestimmten Schlüssel überprüft. Besitzen sie den passenden Schlüssel, werden sie ausgegraut. Im gleichen Zuge werden alle Nachrichten die eine EAnnotation mit entsprechenden Schlüssel besitzen bei der Erstellung der Icons nicht mit berücksichtigt.

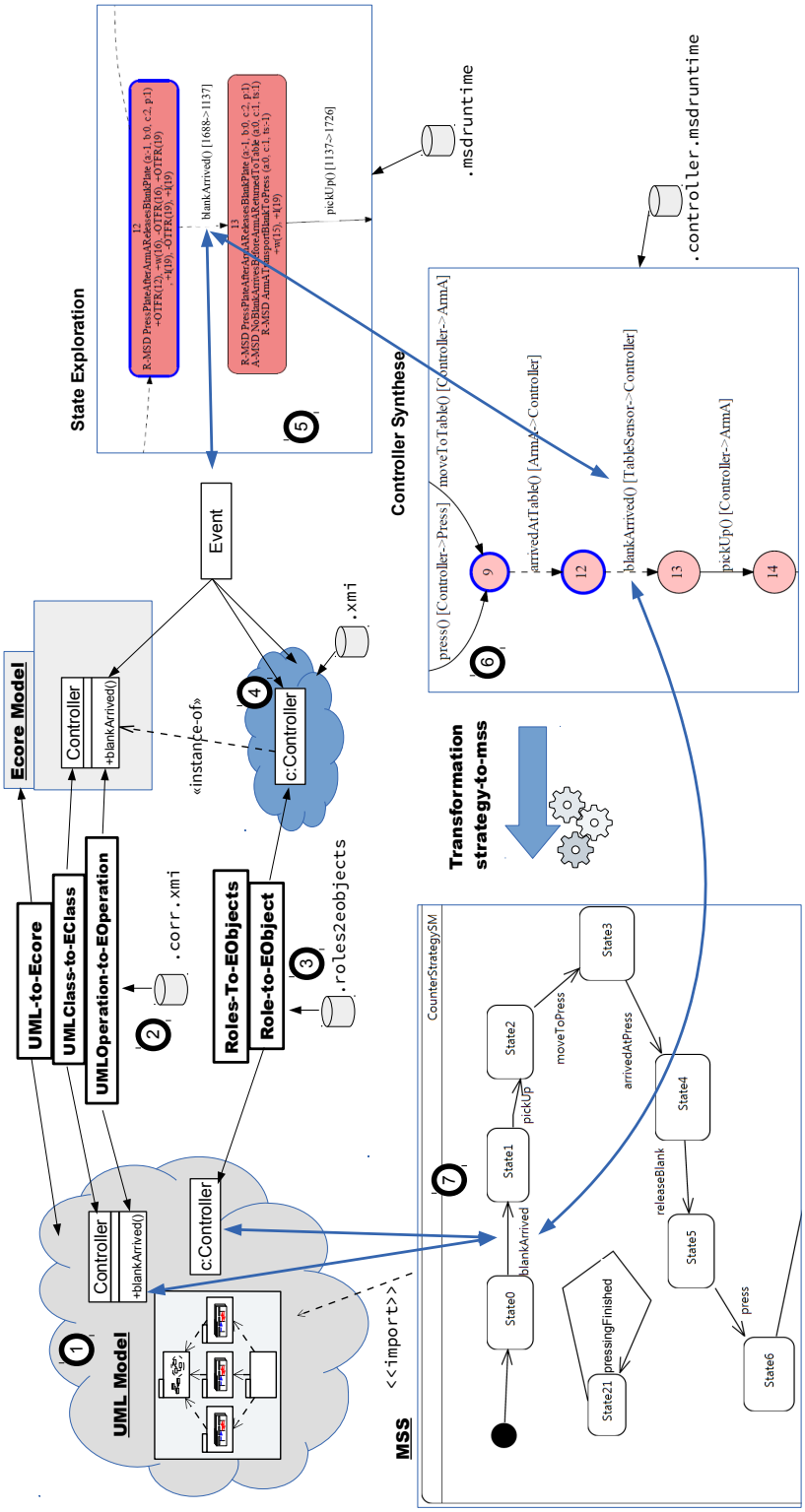


Abbildung 4.2.: Zu sehen sind die einzelnen Komponenten von ScenarioTools, die über eine Nachricht in Bezug gesetzt wurden. Eine Nachricht besteht im Wesentlichen aus einem Funktionsnamen und einem empfangenden Objekt.

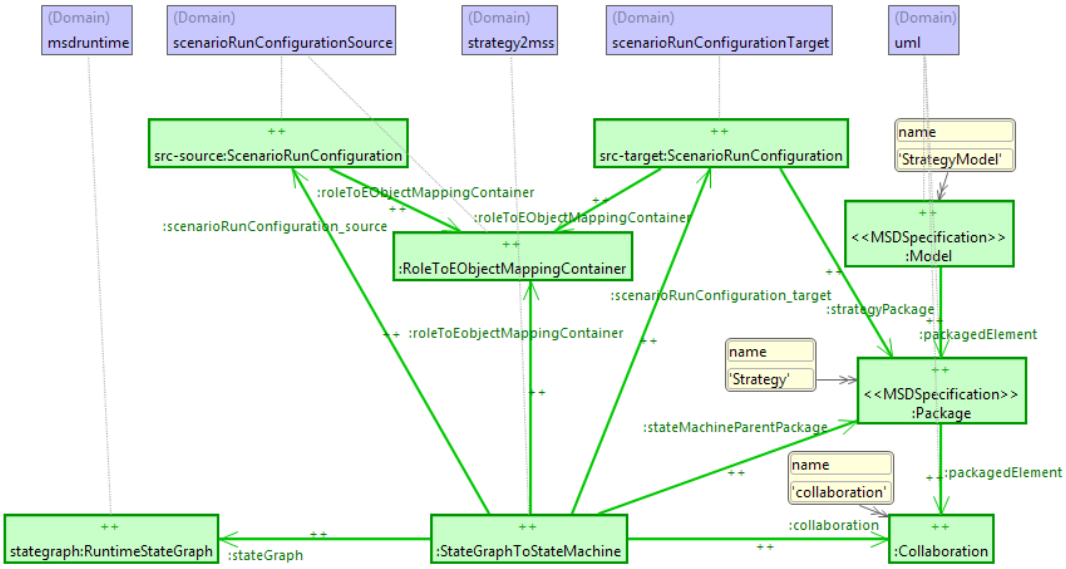


Abbildung 4.3.: Das TGG Axiom der strategy2mss Transformation

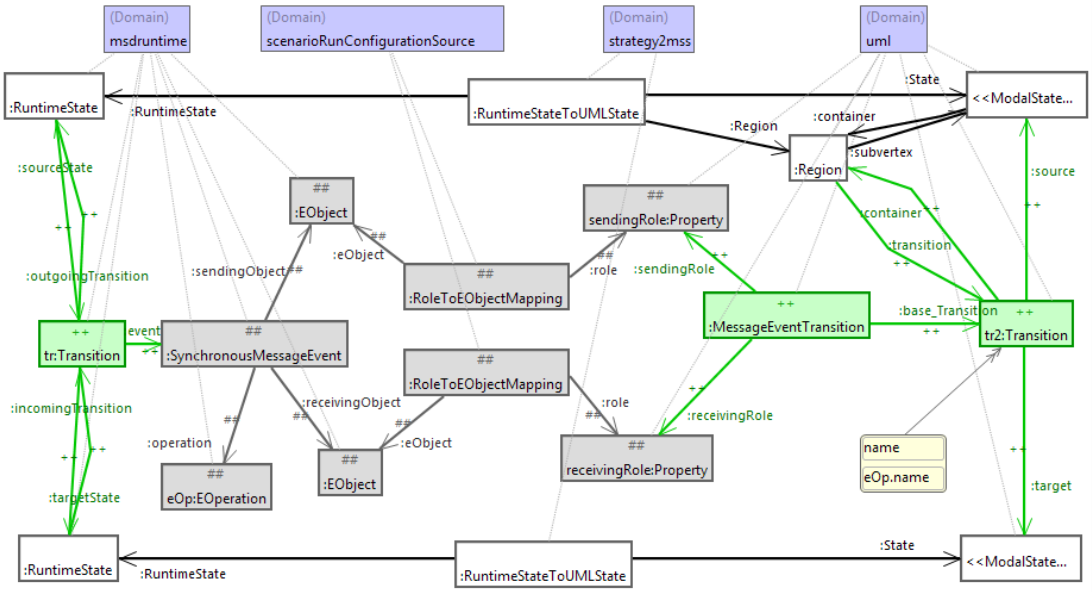


Abbildung 4.4.: TGG Regel, mit der die Zuordnung der Nachrichten umgesetzt wird

5. Implementierung

Die Transformation habe ich in ScenarioTools integriert und wird nun nach einer Benutzerbestätigung automatisch nach der Synthese ausgeführt. Die Synthese wird ausgeführt indem auf einer ScenarioRunConfiguration-Datei das Kontextmenü aufgerufen und *On-The-Fly Synthesis* ausgewählt wird.

5.1. Transformation

Die Strategie zur MSS Transformation habe ich als Plug-In realisiert. Dabei sind die TGG Regeln wie im Entwurf beschrieben übernommen. Zusätzlich kann die Interpreter-Konfiguration von einer Funktion des Plug-Ins automatisch für bestimmte Eingabeparameter erzeugt werden. Die Eingabeparameter sind die Quelldomänen: Die ScenarioRunConfiguration und die Strategie.

Während der Implementierung der Transformation hat sich heraus gestellt, dass für eine automatische Transformation, bei der nicht zwischen Systemstrategie und Gegenstrategie unterscheiden werden muss, diese Information trotzdem vorhanden sein muss. Aus diesem Grund habe ich die Datenstruktur der Strategie um eine Eigenschaft erweitert, die diese Information enthält. Damit ist gewährleistet, dass die Transformation stabil funktioniert bzw. muss auf diese Weise die Interpreter-Konfiguration nicht versteckt werden, da sie alle Informationen enthält. Ein Übergeben dieser Information nach der Synthese wäre eine flüchtige Information und würde bei einer erneuten Ausführung der Transformation, ohne Synthese, ein verfälschtes Ergebnis liefern.

Die Strategie soll alle Nachrichten, die in der Simulation auftreten können, beinhaltet. Um dieses Ziel zu erreichen wurde die Synthese dahingehend erweitert, dass die Strategie einen Zustand enthält, der alle Nachrichten der Zustandsraumerkundung ohne die Nachrichten der Strategie aufnimmt.

5.2. ScenarioRunConfiguration

In der Implementierung habe ich mich dafür entschieden, die ScenarioRunConfiguration um ein Attribut zu erweitern und damit die Strategie in die Simulation zu integrieren. Bei einer normalen Simulation bleibt dieses Feld leer. Beim Ausführen der Transformation, wird hier die Strategie eingetragen. Wird nun eine ScenarioRunConfigurations-Datei geladen, wird in der Initialisierungsmehtode überprüft, ob eine Strategie vorhanden ist.

Liegt dieser Fall nicht vor, wird weiter fortgeschritten. Gibt es eine Strategie, wird sie initialisiert und anschließend mit der Initialisierung der MSD Spezifikation fortgefahren.

6. Verwandte Arbeiten

Die vorliegende Arbeit befasste sich mit der Simulation von unrealisierbaren szenariobasierten Spezifikationen, unter der Zielzielsetzung, den Entwickler bei der Analyse dieser zu unterstützen.
Eine thematisch sehr ähnliche Arbeit haben Maoz und Sa'ar in „Executing Unrealizable Scenario-Based Specifications“ [MS13] verfasst. Dabei integrieren sie ein Counter-Play-Out in ihre Entwicklungsumgebung PlayGo [PlayGoHP]. Mit Hilfe von PlayGo können dann szenariobasierte Spezifikationen auf Basis von LSCs entwickelt und analysiert werden.
Viele andere Arbeiten beschäftigen sich zwar mit der Synthese von realisierbaren Spezifikationen, wie mit unrealisierbaren Spezifikationen verfahren werden soll, bleibt jedoch offen. Diese Problematik ist auch in der Arbeit von Maoz und Sa'ar im Abschnitt *Related Work* ausführlich beschieben.
Ein weiterer interessanter Ansatz, der sich mit der Analyse von Spezifikationen beschäftigt, ist in UPPAAL TIGA [BDL04] integriert. Der Ansatz konzentriert sich auf die Auswertung von Aussagen der *Modallogik*, die sich an die Spezifikation richten. Dabei können gezielte Anfragen gestellt werden, aus denen sich ableiten lässt ob ein System bestimmte Zustände erreicht bzw. ob es beliebig oft in bestimmte Zustände gelangt.

7. Zusammenfassung

Ziel dieser Arbeit war es den Benutzer besser bei der Analyse von unrealisierbaren scenariobasierten Spezifikation zu unterstützen. Dazu bietet ScenarioTools einen Synthese-Algorithmus, der Widersprüche in szenariobasierten Spezifikationen erkennen kann. Das Ergebnis dieses Algorithmus ist ein Graph, der eine Gegenstrategie enthält, die dem Benutzer einen Weg zeigt, wie die Umwelt die Spezifikation verletzen kann. Bei umfangreichen Spezifikationen ist dieser Graph jedoch nicht mehr einfach zu interpretieren. Aus diesem Grund soll die Unterstützung von dem Finden des Fehlers hin zum besseren Verstehen des Fehlers ausgeweitet werden. Dazu sollte die Gegenstrategie in die Simulation integriert werden, um so ein Spiel zwischen Umwelt und Benutzer zu inszenieren, bei dem die Umwelt das System systematisch zu einen Widerspruch führt. Sobald der Benutzer verliert, erkennt er an welcher Stelle seine Spezifikation eine Schwachstelle enthält und kann sie ausbessern.

Um dies umzusetzen habe ich das Counter-Play-Out in ScenarioTools implementiert. Dabei habe ich das Ergebnis der Synthese in das Format der MSD Spezifikation transformiert und die Simulation so angepasst, dass beide zusammen ausgeführt werden können. Die Eigenschaft der dualen Arbeitsweise der Synthese, die sowohl bei realisierbaren als auch bei unrealisierbaren Spezifikation eine (Gegen-)Strategie liefert, konnte bei dieser Arbeit zum Vorteil genutzt werden. So konnten, aufbauend auf dem Syntheseergebnis, mit einer Transformation zwei Use Cases abgedeckt werden: Die Simulation und Ausführung einer Gegenstrategie sowie einer Systemstrategie.

Im Entwurf habe ich einen Vorschlag gemacht in welcher Weise die Benutzeroberfläche intuitiver gestaltet werden könnte. Diesen Vorschlag konnte ich in dieser Arbeit nicht mehr umsetzen. Allerdings kann bei der Simulation auf den Namen des MSS geachtet werden. Dieser gibt Aufschluss darüber, welcher Strategietyp enthalten ist.

A. Anhang

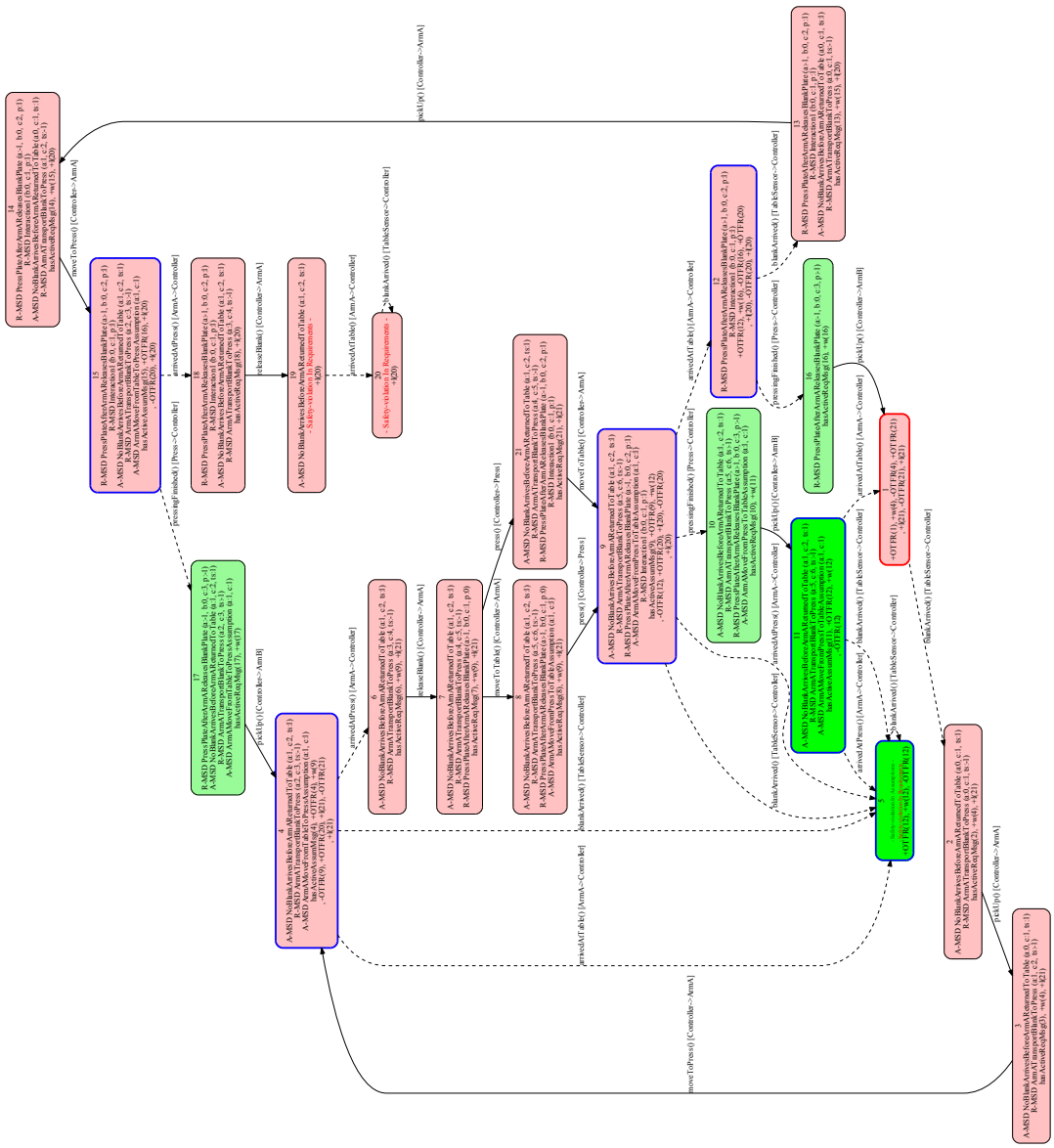


Abbildung A.1.: Ergebnis der Zustandsraumerkundung der *On-The-Fly-Synthese*. Grundlage war das Produktionszellen-Beispiel ohne Annahme 4.

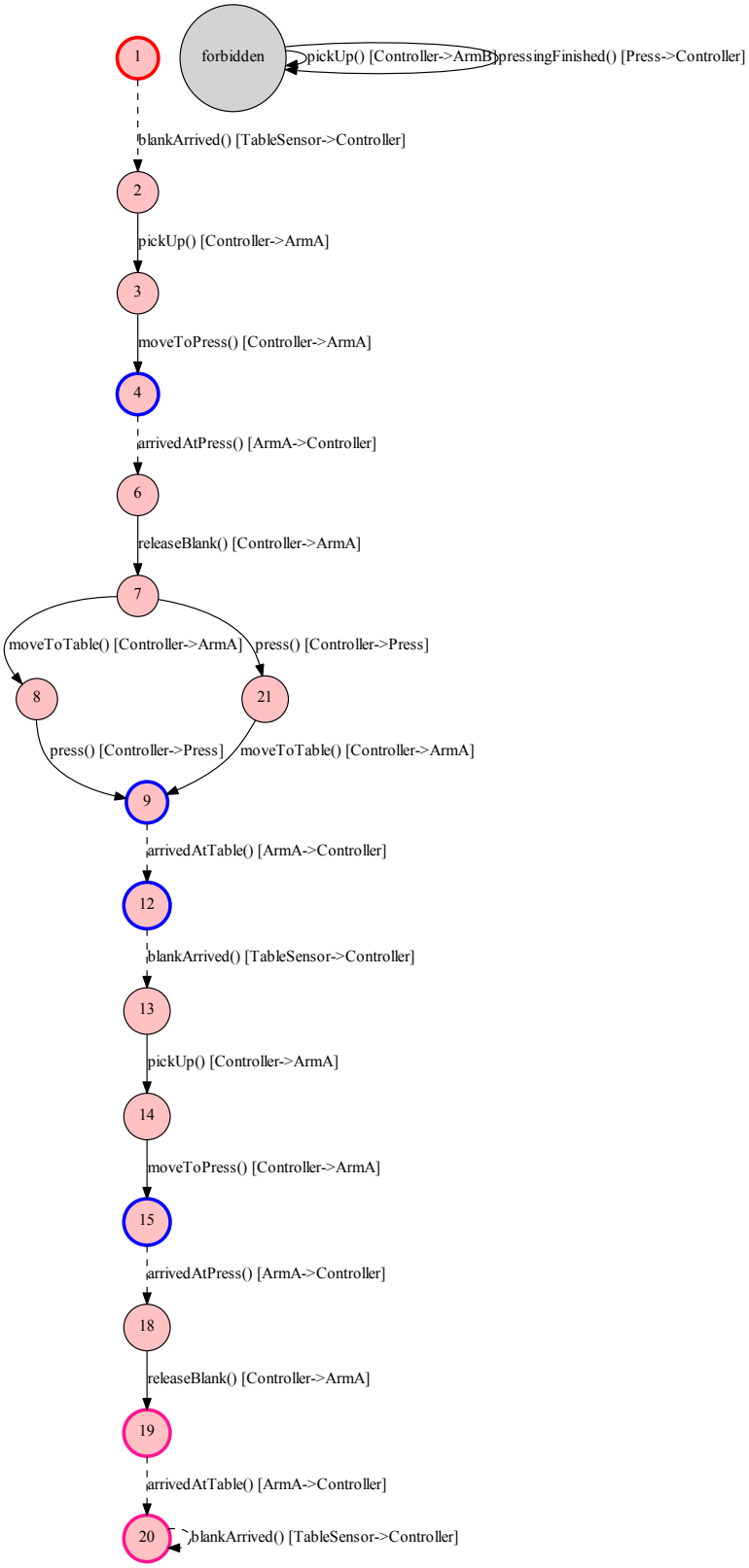


Abbildung A.2.: Ergebnis der Synthese des Produktionszellen-Beispiel ohne Annahme 4.

B. Plug-In SVN Revisionen

Im Folgenden sind die Plug-Ins gelistet an denen ich im Rahmen dieser Arbeit Änderungen vorgenommen habe. Die Revisionnummer steht für die jeweilige Version aus dem zugehörigen SVN Repository. Die Versionen der Plug-ins stellen den aktuellen Stand zum Zeitpunkt der Abgabe dar. Alternativ zur Revisionsnummer kann auch das Datum 4.3.2014 gewählt werden.

Plug-In	**Revision**
org.scenariotools.stategraph	10978
org.scenariotools.stategraph.edit	6507
org.scenariotools.msd.runtime	12059
org.scenariotools.msd.runtime.edit	11523
org.scenariotools.msd.runtime.editor	11221
org.scenariotools.msd.strategy2mss	12170
org.scenariotools.msd.synthesis.otfb.ui	11873

Literaturverzeichnis

[Come, Let's Play] David Harel, and Rami Marelly,
Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine,
Springer-Verlag, 2003

[G11] Joel Greenyer,
Scenario-based design of mechatronic systems,
Dissertation, 2011

[BGP13] Christian Brenner, Joel Greenyer, and Valerio Panzica La Manna,
Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013) The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions,
Electronic Communications of the EASST, 2013

[BGPGS13] Christian Brenner, Joel Greenyer, Valerio Panzica La Manna, Carlo Ghezzi and Wilhelm Schäfer,
ScenarioTools: Simulation and Synthesis for Rich Modal Scenario Specifications,
Review Version, 2013

[G13] Joel Greenyer,
Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications - Seminar Talk at the Weizmann Institute,
Seminar Talk, 2013

[STHP] ScenarioTools Projekt Homepage,
http://scenariotools.org/,
letzer Zugrif : März 2014

[G06] Joel Greenyer,
A Study of Model Transformation Technologies: Reconciling TGGs with QVT,
Diplomarbeit, 2006

[GR12] Joel Greenyer, and Jan Rieke,
Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata,
Springer-Verlag, 2012

[MS13] Shahar Maoz, and Yaniv Sa'ar,
Counter Play-out: Executing Unrealizable Scenario-based Specifications,
IEEE Press, 2013

[PlayGoHP] PlayGo Wiki Main Page,
http://www.weizmann.ac.il/mediawiki/playgo/index.php/Main_ Page,
letzer Zugrif : März 2014

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen,
Tutorial on UPPAAL,
Springer-Verlag, 2004, pp 200-236

Abbildungsverzeichnis

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 04.03.2014

_____

Timo Gutjahr