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

Generierung von Code zur Ausführung und Zustandsraumexploration von multimodalen szenariobasierten Spezifikationen

Bachelorarbeit

im Studiengang Informatik

von

Nils Glade

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

Hannover, 16.09.2014

ABSTRACT

Reactive (software) systems often consist of, sometimes many, distributed components, that have to interact with each other and their environment. This can lead to complex behaviour patterns that must be captured by the systems specification. A feasible approach to this problem are scenario based specifications. Depending on the notation and used formalism, these specifications can be executed and checked for inconsistency. In this thesis an existing software for creating scenario-based specifications is extended with the capability of executing code, that has been generated from said specifications. In order to show the expressive power of the generated code, a transformation of the essential features of modal sequence diagrams into java code is given.

ZUSAMMENFASSUNG

Reaktive (Software-)Systeme bestehen oft aus vielen (verteilten) Komponenten, die untereinander in komplexen Mustern interagieren müssen. Eine Möglichkeit das Verhalten dieser Systeme zu beschreiben sind szenariobasierte Spezifikationen. Abhängig von der verwendeten Notation ist es möglich, diese Spezifikationen auszuführen und auf Widerspruchsfreiheit zu testen. In dieser Arbeit wird eine bestehende Software zur Erstellung und Test szenariobasierter Spezifikationen um die Möglichkeit erweitert, Code auszuführen, der aus diesen Spezfikationen erzeugt werden kann. Für die wichtigsten Features von modalen Sequenzdiagrammen wird angegeben, wie aus diesen Code generiert werden kann.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

ACRONYMS

MSD Modales Sequenzdiagramm

MSS Modal State Structure

UML Unified Modeling Language

EMF Eclipse Modeling Framework

OCL Object Constraint Language

1

EINLEITUNG

1.1 MOTIVATION

Die Entwicklung von reaktiven Systemen ist ein fehleranfälliger Prozess. Die Gründe hierfür liegen zum Einen in der zunehmenden Verwendung verteilter Systeme, d.h. Systeme deren Funktionalität sich durch die Zusammenarbeit mehrerer diskreter Komponenten ergibt. Dieser Trend wird z.B. an der Entwicklung von Automobilen deutlich: nach Pretschner et al. gibt es moderne Autos, die über dutzende von integrierten Prozessoren verfügen und hunderte verschiedene Funktionen ermöglichen [PBKS07].

Zum Anderen ist es oft auch für System mit weniger Komponenten schwierig, das *vollständige* Systemverhalten korrekt und präzise zu beschreiben. Ein Lösungsansatz besteht in der Spezifikation des Systemverhaltens mittels Szenarien, die die Interaktion des Systems mit seiner Umgebung bzw. den Benutzern beschreiben. Es ist jedoch auch wichtig, diese Spezifikation so früh wie möglich auf Widerspruchsfreiheit, beziehungsweise Korrektheit zu prüfen. Wird gegen Ende eines Projekts festgestellt, das die Spezifikation fehlerhaft war, müssen Teile, oder schlimmstenfalls das komplette System neu implementiert werden, wodurch der Projekterfolg gefährdet ist. Eine präzise Notation des Systemverhaltens in Szenarioform ist zum Beispiel mittels *modaler Sequenzdiagramme* (MSD) möglich. Diese sind eine Variante der häufig genutzten Sequenzdiagramme der Unified Modeling Language (UML). Durch die im Vergleich zu normalen Sequenzdiagrammen genaueren Semantik ist es möglich, MSD-Spezifikationen mittels des *Playout-Algorithmus* auszuführen. Desweiteren kann durch (automatische) Analyse des Playout-Verhaltens gezeigt werden, ob eine Spezifikation realisierbar ist. Die Software ScenarioTools ermöglicht das Erstellen von MSD-Spezifikationen, deren Simulation mit dem Playout-Algorithmus, sowie die Prüfung von erstellten Spezifikationen auf Realisierbarkeit. Das Problem ist, das die Software nicht in der Lage ist, Szenarien, bzw. Spezifikationen zu verarbeiten, die in anderen Sprachen/Notationen verfasst sind.

1.2 ZIELE DIESER ARBEIT

In dieser Arbeit soll ScenarioTools so modifiziert werden, dass die Ausführung beliebiger szenariobasierter Spezifikationen ermöglicht wird. Dazu soll die für die Ausführung von Spezifikationen zuständige Komponente (*Runtime*) neu entwickelt werden. Anstatt Spezifikationen zu interpretieren, soll Code von der Runtime ausgeführt werden können, der aus den Szenarien der Spezifikation generiert werden kann.

CODE REUSE Für die neue Implementierung sollte, Code aus der alten Runtime wiederverwendet werden, sofern dies sinnvoll ist. Durch Beibehaltung existierender Schnittstellen muss Code, der von der Runtime abhängig ist, nur leicht, oder im besten Fall überhaupt nicht angepasst werden.

ERWEITERBARKEIT Um die Integration neuer Sprachen oder Formalismen in ScenarioToolszu erleichtern, sollte die neue Runtime und insbesondere der generierte Code leicht erweiterbar sein. Dazu sollte die Schnittstelle zwischen Runtime und generiertem Code möglichst klein sein. Außerdem sollten so wenig Annahmen wie möglich über die Form/Struktur des generierten Codes getroffen werden.

1.3 STRUKTUR DER ARBEIT

Diese Arbeit ist in sieben Kapitel unterteilt. In Kapitel 1 wird das Thema und Ziel der Arbeit vorgestellt. In Kapitel 2 werden die notwendigen Grundlagen des Themas, d.h. die Definition reaktiver Systeme, Erläuterung modaler Sequenzdiagramme, und die Software ScenarioToolserläutert. In Kapitel 3 wird deren Implementierung untersucht, Kapitel 4 enthält einen Vergleich zwischen der in dieser Arbeit neu entwickelten und der bereits vorhandenen Implementierung. Kapitel 5 beschreibt, wie der Zustandsraum einer gegebenen Spezifikation untersucht werden kann. In Kapitel 6 werden verwandte Arbeiten vorgestellt. Kapitel 7 enthält eine Zusammenfassung der Ergebnisse dieser Arbeit sowie mögliche Aspekte die in Zukunft erweitert werden könnten.

2

GRUNDLAGEN

In diesem Kapitel werden grundlegende Konzepte erklärt, die für das weitere Verständnis dieser Arbeit wichtig sind.

2.1 REAKTIVE SYSTEME

Der Begriff *reactive system* wurde 1985 von Harel und Pnueli zur Beschreibung einer bestimmten Art von (Software-)Systemen eingeführt [HP85]. Ein reaktives System steht in ständigem Kontakt mit seiner Umgebung und muss auf Nachrichten, bzw. Signale dieser Umgebung reagieren.

2.2 UML SEQUENZDIAGRAMME

Die UML ist eine von der Object Modeling Group standardisierte grafische Modellierungssprache [OMG11]. Sie enthält verschiedene Arten von Diagrammen zur Modellierung von Struktur und Verhalten von (Software-)Systemen. Szenariobasierte Spezifikationen können in UML mithilfe von Sequenzdiagrammen erstellt werden. Ein Sequenzdiagramm stellt eine Interaktion mehrerer Objekte dar. Die Kommunikation dieser Objekte wird durch das Senden und Empfangen von Nachrichten modelliert.

Ein einfaches Beispiel ist in Abbildung 2.1 dargestellt. Das Sequenzdiagramm *BuyBeverage* spezifiziert den Bezahlvorgang an einem Getränkeautomaten (*VendingMachine*). Der Kunde *c* wählt zunächst ein Getränk aus (*selectBeverage(b)*). Anschließend muss er solange Münzen in den Automaten eingeben (*insertCoin()*), bis das Getränk vollständig bezahlt wurde (*sum* < *price*). Danach wird das Getränk ausgegeben *dispense(b)*.

Kunde und Automat werden als Objekte modelliert, die im Sequenzdiagramm an *Lifelines* (gestrichelte vertikale Linien) gebunden werden. Das Rechteck über einer Lifeline enthält Informationen darüber, welche Objekte an diese gebunden werden können. Zum Beispiel wird für das Objekt *c* festgelegt, das es zur Klasse *Customer* gehören muss, die in einem anderen Teil der Spezifikation beschrieben werden kann. Ein Pfeil zwischen zwei Lifelines repräsentiert eine Nachricht, die von einem Objekt an ein anderes gesendet wird und

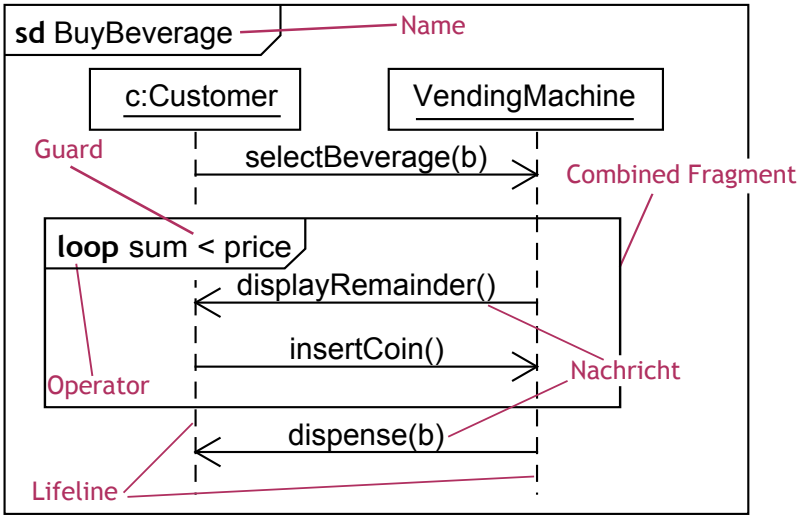

Abbildung 2.1: Bezahlvorgang an einem Getränkeautomaten

ist mit dem Namen der Nachricht und, falls die Nachricht parametrisiert ist, einer Liste von Parametern in Klammern beschriftet. Der Parameter *b* z.B. könnte für das gewählte Getränk stehen. Die Reihenfolge in der Nachrichten gesendet werden sollen wird durch die Position der Nachricht auf der sendenden und empfangenden Lifeline bestimmt. Zum Beispiel liegt *selectBeverage* „über“*displayRemainder*, d.h. *selectBeverage* sollte früher gesendet werden.

Da der Automat das Getränk erst ausgeben sollte, wenn der komplette Preis bezahlt wurde, werden die Nachrichten *displayRemainder()* und *insertCoin()* mit einem *Combined Fragment* umgeben, das eine wiederholte Ausführung (*loop*) spezifiziert. Neben einer Bedingung (*Guard*) wie *sum* < *price* können auch Zahlen angegeben werden, um z.B. zu erzwingen, das der Inhalt *(Interaction Operand)* des Fragments mindestens oder nicht öfter als eine bestimmte Anzahl passieren muss.

Außer *loop* gibt es noch weitere Operatoren (*InteractionOperator*). Ein Fragment mit dem *par* Operator signalisiert, dass mehrere Interaktionen parallel stattfinden. Sind zu einem bestimmten Zeitpunkt verschiedene Abläufe möglich kann dies mit einem *alt*-Fragment ausgedrückt werden. Aus den Operanden des Fragments wird höchstens eine Alternative ausgewählt. Die Verwendung von *alt* und *par* wird in Abbildung 2.2 illustriert. Das Sequenzdiagramm spezifiziert die Anforderung „Es darf kein Bier an Kinder verkauft werden“. Wenn der Kunde Bier als Getränk auswählt, fragt der Automat nach dem Ausweis des Kunden, um dessen Alter auszulesen. Der erste Operand des *alt*-Fragments wird nur betreten, falls der Kunde alt genug ist (*c.age* > *16*). Das *par*-Fragment innerhalb des ersten Operanden gibt an, das gleichzeitig (*„parallel“*) das Getränk und eventuell vorhandenes Wechselgeld ausgegeben werden soll.

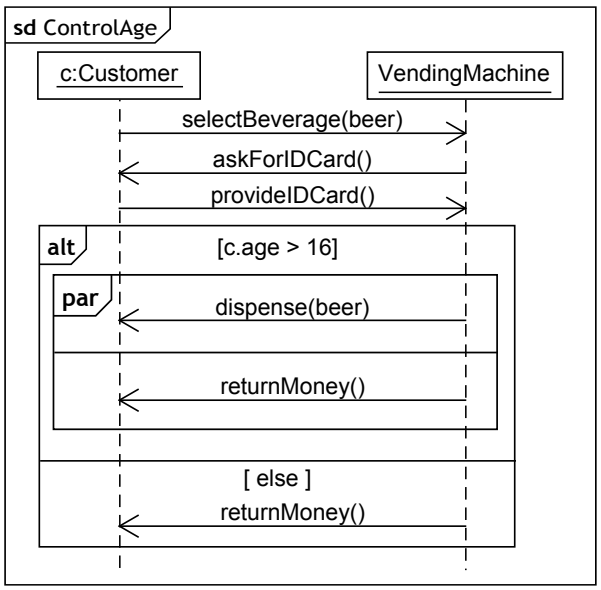

Abbildung 2.2: Beispiel für *alt* und *par*-Fragmente

2.3 MODALE SEQUENZDIAGRAMME

Modale Sequenzdiagramme wurden erstmalig von Harel und Maoz beschrieben [HM08]. In einem modalen Sequenzdiagramm können Diagrammelemente wie Nachrichten, Fragmente oder Invarianten mit *Modalitäten* versehen werden. Mit diesen Modalitäten können *Liveness* und *Safety*-Eigenschaften des spezifizierten Systems festgelegt werden. Die informelle Bedeutung der beiden Begriffe ist, das „nichts schlechtes“ passieren wird (*safety*), und das „gute Dinge“ irgendwann passieren (*liveness*) (Zitat von Lamport, nach Alpern und Schneider [AS87]).

2.3.1 *Modalitäten*

Harel und Maoz definieren eine Modalität Temperatur (*temperature*). Die Temperatur kann entweder *hot*, oder *cold* sein. Die Modalität wird einerseits durch die Farbe der Nachricht (hot/cold), andererseits durch die Strichelung der Pfeile (hot: durchgezogen, cold: gestrichelt) dargestellt. Die Modalitäten können auch auf komplette Diagramme übertragen werden: *universal sequence diagrams* enthalten Szenarien, die während jeder Ausführung (*run*) geschehen müssen; *existential sequence diagrams* enthalten Szenarien, die in mindestens einem Run erfüllt werden. In Abbildung 2.3 sind zwei einfache Modale Sequenzdiagramme (MSDs) dargestellt. Im linken Diagramm ist die Nachricht *m1* cold, die Nachricht *m2* hot und die Nachricht *m3* wieder cold. Das Diagramm besagt, dass in jedem Run nachdem m1 gesendet wurde, weder m1 noch m3 gesendet werden dürfen, bis m2 gesendet wurde,

und das es möglich ist, das nach m1 und m2 auch m3 gesendet wird. Brenner et al. definieren zusätzlich zur Temperatur die Modalität *execution kind* [BGP13]. Ist eine Nachricht *executed*, *muss* sie irgendwann gesendet werden. Ist sie hingegen *monitored*, werden auch Runs erlaubt, in denen die Nachricht niemals auftritt. Durchgezogene Pfeile stellen Nachrichten dar, die *executed* sind. Gestrichelte Pfeile stellen Nachrichten dar, die *monitored* sind. Zum Beispiel ist die Nachricht m7 im Diagramm seq2 (Abbildung 2.3) executed, die Nachrichten m6 und m8 hingegegen monitored.

liveness

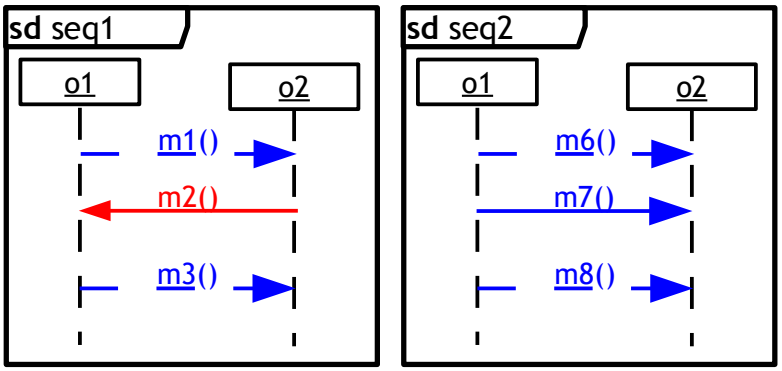

Abbildung 2.3: Zwei Modale Sequenzdiagramme

2.3.2 *Symbolische Nachrichten und Lifelines*

Wenn verschiedene Objekte an eine Lifeline gebunden werden können, die eine bestimmte Bedingung erfüllen sollen, kann dies mit symbolischen Lifelines ausgedrückt werden. Eine symbolische Lifeline wird mit einem Ausdruck der Object Constraint Language (OCL) versehen [BGP13]. In Abbildung 2.4 ist dies beispielhaft dargestellt. An die rechte Lifeline können nur Objekte gebunden werden, die den Ausdruck o1.getO2() erfüllen. Um eine Nachricht zu modellieren,

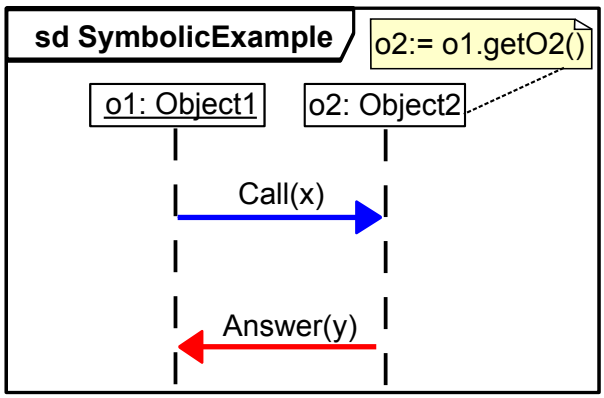

Abbildung 2.4: Modales Sequenzdiagramm mit symbolischen Lifelines und symbolischen Nachrichten

die Parameter besitzt, deren Wert aber nicht direkt angegeben werden soll,können symbolische Nachrichten verwendet werden. Eine symbolische Nachricht enthält statt konkreter Parameter wie 5 oder „Mary“ Variablen wie *amount* oder *name*.

2.3.3 *Invarianten*

Um festzulegen, das eine bestimmte Bedingung zu einem bestimmten Zeitpunkt gelten soll, können *state invariants* benutzt werden. Diese werden als abgerundete Rechtecke auf eine Lifeline gezeichnet. Die Bedingung die gelten soll, wird in das Rechteck hineingeschrieben.

2.3.4 *Unterscheidung von Annahmen und Requirements*

Brenner et al. schlagen für die Modellierung von reaktiven Systemen die Trennung von MSDs, die Anforderungen an das System beschreiben (*Requirement MSDs*) und MSDs die Annahmen über die Umwelt des Systems beschreiben (*Assumption MSDs*) [BGP13].

2.4 PLAY-OUT

Modale Sequenzdiagramme können die Anforderungen an das Systemverhalten genauer beschreiben als normale Sequenzdiagramme. Dennoch kann eine MSD-Spezifikation inkonsistent oder fehlerhaft sein, z.B. wenn Modalitäten falsch zugewiesen wurden, oder falsche, beziehungsweise optimistische Annahmen getroffen wurden. Mit dem Play-Out-Algorithmus ist es möglich, eine Spezifikation auszuführen um zu überprüfen, ob das Systemverhalten dem erwarteten Verhalten entspricht. Der Algorithmus wurde ursprünglich von Harel & Marelly für *Live Sequence Charts*, einem den modalen Sequenzdiagrammen ähnlichen Formalismus vorgestellt [HM03], kann aber auch auf MSD-Spezifikationen angewandt werden. Seine Funktionsweise wird im Folgenden anhand einer Beispielspezifikation erklärt.

Abbildung 2.5 enthält einen Teil einer Spezifikation einer automatisierten Fabrik (ProductionCell). Das Beispiel wurde aus einem Artikel von Greenyer et al. [GBC+13] entnommen. Auf einem Förderband werden Metallrohlinge zu einem Wartebereich transportiert.

Wenn ein Rohling im Wartebereich angelangt, wird ein Steuergerät (*Controller*) über einen Sensor (*TableSensor*) benachrichtigt. der Controller aktiviert einen Roboterarm (*ArmA*), der den Rohling auf einer Presse (*Press*) ablegt. Nachdem der Rohling gepresst wurde, weist der Controller einen anderen Arm (*ArmB*) an, den Rohling auf ein anderes Förderband abzulegen. Eine Illustration des Beispiels ist in dem Artikel von Greenyer et al. [GBC+13] und auf der Internetseite

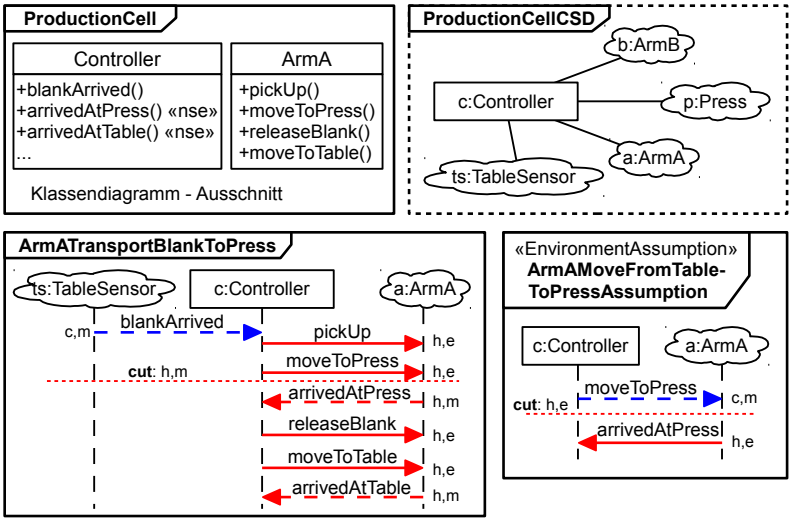

Abbildung 2.5: PRODUCTIONCELL-Spezifikation. Oben links: Klassendiagramm. Oben rechts: Kollaboration der einzelnen Objekte. Unten: aktive Diagramme

von SCENARIOTOOLS[1] (siehe auch Abschnitt 2.5) zu finden. Das obere rechte Diagramm beschreibt eine Kollaboration von mehrerer Rollen(*roles*). Konkrete Objekte können an eine Rolle *gebunden* werden, wenn sie den gleichen Typ wie diese Rolle haben. Lifelines in einem Sequenzdiagramm können sich auf diese Rollen beziehen. Die Wolkenform der Rollen a,b,p,ts deutet an, das diese Rollen das Environment darstellen.

Für die Ausführung werden konkrete Objekte benötigt, die an die vorgegebenen Rollen gebunden werden können (Instanzenmodell). Die Interaktion dieser Objekte wird vom Playout-Algorithmus folgendermaßen simuliert: In jedem Schritt wird eine Nachricht ((*message-*)*event*) gesendet, wobei die Annahme getroffen wird, das das System unendlich viel schneller agieren kann als das Environment. Nachrichten von Systemobjekten werden daher immer vor Nachrichten von *environment*-Objekten gesendet. Die Korrektheit dieses Schrittes, d.h. ob die Nachricht der Spezifikation zufolge gesendet werden darf, wird durch das Erstellen und Kontrollieren aktiver Kopien (*active copy*, aktives Diagramm) überprüft.

Eine aktive Kopie eines Diagramms wird dazu benutzt, zu verfolgen, ob Nachrichten, die in diesem Diagramm notiert sind (*diagram message*), in der richtigen Reihenfolge auftreten. Der *cut* einer aktiven Kopie bestimmt die *diagram messages*, die zu einem bestimmten Zeitpunkt erlaubt sind (*enabled*). Er wird durch eine, bzw. mehrere

1 http://scenariotools.org/what-is-scenariotools/msd-specifications/ abgerufen am 14.09.2014

horizontale Linien dargestellt (siehe z.B. Abbildung 2.5 unten links), die jede Lifeline des Diagramms schneidet. Der cut verfügt ebenso wie *diagram messages* über eine Modalität, die von den Modalitäten der *diagram messsages* vor (unter) dem cut, bestimmt wird. Wenn sich mindestens eine *diagram message* die hot ist, vor dem cut befindet, ist dieser auch hot, andernfalls ist er cold. Analoges gilt für *monitored* und *executed*. Zum Beispiel befindet sich der cut in beiden Sequenzdiagrammen in Abbildung 2.5 vor der *diagram message arrivedAtPress*. Diese ist in beiden Diagrammen hot, also sind auch beide cuts hot. Im linken Diagramm ist allerdings *arrivedAtPress* nur monitored, also ist auch der linke cut monitored.

Wenn während der Simulation ein *message event* gesendet wird kann folgendes passieren:

– Wenn das *event unifizierbar* mit einer *enabled diagram message* in einem aktiven Diagramm ist, wird der cut hinter die *diagram message* verschoben mit der das *event* unifiziert wurde. Ein *event* ist mit einer *diagram message* unifizierbar, wenn es den gleichen Sender, den gleichen Empfänger, und den gleichen Namen wie die *diagram message* hat. Ist die *diagram message* parametrisiert, ist das *event parameter-unifizierbar*, wenn alle Werte der Parameter gleich sind. Ein *event* ist mit einer symbolischen Nachricht parameter-unifizierbar, wenn die symbolischen Parameter vom gleichen Typ sind, wie die konkreten Parameter.

–Wenn das Event unifizierbar mit einer Nachricht ist, die in einem aktiven Diagramm vorkommt, aber nicht enabled ist, wurde das Diagramm verletzt (*violation*). Ist der cut zu diesem Zeitpunkt hot, führt dies zu einer *safety violation*, ist der cut cold, führt dies zu einer *cold violation*. Im ersten Fall wurden, abhängig vom Diagrammtyp (Assumption MSD, Requirement MSD), die Annahmen über die Umwelt bzw. die Anforderungen an das System verletzt. Im zweiten Fall wird lediglich das aktive Diagramm verworfen.

–Wenn das Event mit einer *minimalen* Nachricht in einem beliebigen Diagramm unifizierbar ist, wird eine aktive Kopie dieses Diagramms erstellt, Objekte an die Lifelines gebunden und der cut hinter die minimale Nachricht verschoben.

minimal≡ es befindet sich keine Nacricht vor dieser Nachricht auf der sendenden/empfangenden Lifeline

–Wenn der cut nur noch eine Nachricht enthält, und diese mit dem Event unifizierbar ist, wird das Diagramm geschlossen.

Während der Simulation wird versucht *violations* zu vermeiden, indem nur Events gewählt werden, die in den derzeit aktiven Diagrammen *enabled* sind. Algorithmus 2.1 beschreibt das Playout-Verhalten in Pseudocode.

Algorithmus 2.1: Play-Out

```
Input : MSD-Specification 𝔐, Instance Model 𝔒
1  while no safety violation occurred do
2      Event e;
3      if System active then
4          e ←choose (non violating) system message ;
5      else
6          e ←choose (non violating) environment message ;
7      foreach Active Diagram d do
8          progresscut(d,e);
9          if d is finished or violation occurred then
10             remove(d);
11     activateDiagrams(e);
12 abort;
```

Um die in Abbildung 2.5 dargestellte Situation zu erreichen, könnte der Algorithmus folgende Schritte durchgeführt haben:

1. Wähle *blankArrived*, aktiviere ArmATransportBlankToPress, verschiebe cut
2. Wähle *pickUp*. Verschiebe cut.
3. Wähle *moveToPress*. Aktiviere ArmA... Assumption, verschiebe cut

2.5 SCENARIOTOOLS

ScenarioTools ist eine Sammlung von Plugins für die freie Entwicklungsumgebung Eclipse. Die Software verfügt über einen UML-Editor, mit dem MSD-Spezifikationen erstellt werden können. Diese können in einer Simulationsumgebung ausgeführt werden, die drei verschiedene PlayOut-Varianten unterstützt. Um die Realisierbarkeit einer Spezifikation zu überprüfen, kann mittels einer Synthese-Komponente eine Steuerung (*Controller*) für das spezifizierte System erzeugt werden. Falls ein Controller synthetisiert werden kann, ist dies ein Beweis für die Realisierbarkeit der Spezifikation.

2.6 ECLIPSE MODELING FRAMEWORK

Das Eclipse Modeling Framework (EMF) ist ein Framework zur Erstellung von Metamodellen in Eclipse. Zur Modellierung dieser Metamodelle wird das *Ecore*-Modell genutzt. In Ecore können Klassen und deren (statische) Beziehungen, ähnlich wie in UML-Klassendiagrammen modelliert werden. Mittels eines *Generator Models* kann aus den modellierten Metamodellen Java-Code generiert werden.

3

IST-ANALYSE VON SCENARIOTOOLS

Das Ziel dieser Arbeit ist es, mithilfe von generiertem Code die Ausführung beliebiger szenariobasierter Spezifikationen mittels Playout oder ähnlichen Algorithmen zu ermöglichen.

Dazu müssen die Daten und Funktionen die dieser Code zur Verfügung stellen muss festgelegt werden (*interface*). Desweiteren muss eine Laufzeitumgebung vorhanden sein, innerhalb derer der generierte Code ausgeführt werden kann. Dies ist jedoch mit der derzeitigen Version von ScenarioTools nicht möglich. Das Programm kann zwar MSD/UML-Spezifikationen mit dem Playout-Algorithmus ausführen, allerdings werden diese nicht in Code übersetzt, sondern interpretiert. Um generierten Code ausführen zu können, muss daher die Laufzeitkomponente (*Runtime*) von ScenarioTools angepasst oder erneuert werden. Um die Neuentwicklung der Runtime zu vereinfachen, wurde zunächst die vorhandene Implementierung daraufhin untersucht, welche Teile sich wiederverwenden lassen, angepasst werden müssen, oder überflüssig sind. Im folgenden wird zunächst kurz die Paketstruktur des Quellcodes beschrieben.

3.1 STRUKTUR

Der ScenarioTools Code ist in fünf Pakete aufgeteilt. Die Klassen in jedem Paket sind einem Ecore-Modell definiert.

EVENTS Das *events* Paket enthält Klassen, die Nachrichten und Ereignisse modellieren. Konkrete Nachrichten werden durch die Klasse *RuntimeMessage* modelliert. Um auszudrücken, das eine Nachricht synchron oder asynchron gesendet wird, muss die *runtime message* in einem *MessageEvent* gekapselt werden. *MessageEvent* und seine Unterklassen sind von der allgemeinen Klasse *Event* abgeleitet. In der derzeitigen Implementierung werden nur synchrone *message events* benutzt.

STATEGRAPH Das *stategraph* Paket enthält Klassen, die einen Zustandsgraphen modellieren. Ein Zustandsgraph (*StateGraph*) besteht aus einer Menge von Zuständen (*State*), die über Transitionen (*Transi-*

tion) verbunden sind. Jede Transition kann mit einem *event* versehen werden.

SYNTHESIS Im *synthesis* Paket sind Algorithmen implementiert, die den Zustandsraum einer Spezifikation untersuchen, und eine Steuerung daraus extrahieren können, falls eine existiert.

RUNTIME / MSDRUNTIME Die Runtime-Pakete enthalten Code und Datenstrukturen, die für die Ausführung des Playout wichtig sind.

3.2 ANALYSE

Eine oberflächliche Betrachtung der Pakete *events*, *runtime* und *stategraph* ergab, dass der Hauptteil des vorhandenen Codes automatisch aus den entsprechenden Ecore-Modellen generiert wurde. Das *synthesis* Paket ist für die Ausführung von Spezifikationen nicht relevant und wurde daher nicht weiter betrachtet.

Das *runtime* Paket enthält die folgenden fünf Klassen: *RuntimeState*, *RuntimeStateGraph*, *ObjectSystem*, *ModalMessageEvent* und *Modality*. Die ersten beiden Klassen sind von den allgemeinen Klassen *State*, *StateGraph* abgeleitet. Sie werden dazu genutzt, den Zustandsraum des Laufzeitverhaltens zu modellieren. Die Klasse *ObjectSystem* modelliert ein Instanzenmodell. Jedes konkrete Objekt wird in eine von zwei Listen eingeteilt, abhängig davon ob es zum *environment* gehört oder nicht. Um modale Nachrichten zu modellieren werden die Klassen *ModalMessageEvent* und *Modality* benutzt. Mit der *Modality*-Klasse besitzt die Attribute *mandatory safety-violating*. Ein *modal message event* kapselt ein normales *message event*. Es verfügt über zwei *modality*-Referenzen um die Modalitäten zu unterscheiden, die das *message event* in *assumption MSDs* und *requirement MSDs* hat. Jede dieser Klassen ist komplett generiert und enthält rein deklarativen Code. Das *runtime* Paket definiert also lediglich eine abstrakte Schnittstelle. Jede der fünf beschriebenen Klassen wird im *msdruntime* Paket erweitert. Dieses wird im Folgenden anhand einer Simulation der PRODUCTIONCELL-Spezifikation näher untersucht.

3.2.1 *Eingabedaten*

Neben der eigentlichen Spezifikation benötigt SCENARIOTOOLSein Instanzenmodell (siehe Abschnitt 2.4) und Informationen über erlaubte Bindungen. Diese müssen angegeben werden, da das Instanzenmodell Instanzen von Ecore-Klassen enthält, die Kollaborationen innerhalb der Spezifikation aber auf UML-Klassen verweist. Für ein statisches System ist von vornherein bekannt, welche Objekte in diesem System zusammenarbeiten. Für die Simulation dieser Systeme muss daher für jede Rolle ein Objekt festgelegt werden, das an die-

role to object binding

se Rolle gebunden wird. Bei der Simulation dynamischer Systeme ist es möglich, das mehrere (dutzende) Objekte an eine Rolle gebunden werden können und das sich das Instanzenmodell während der Ausführung ändert. Anstatt ein Binding für jedes Objekt anzugeben, wird jeder UML-Klasse der Spezifikation eine Ecore-Klasse zugeordnet. Diese Daten können aus einer vorher erstellten Konfigurationsdatei (*scenariorunconfiguration*) geladen werden. Für das PRODUCTIONCELL-Beispiel verwenden wir folgendes Instanzenmodell: $\mathcal{O} = \{ts, c, a, b, p\}$. Die Objekte wurden nach den Rollen der Kollaboration benannt. Jeder Rolle wird das gleichnamige Objekt zugeordnet: $ts \mapsto ts, c \mapsto c$, usw.

uml to ecore

3.2.2 *Simulationsmodell*

Um MSD-Spezifikationen ausführen zu können, werden die Klassen *RuntimeState* und *RuntimeStateGraph* erweitert. Jeder *runtime state* setzt sich aus folgenden Teilen zusammen:

EVENT-MAP Diese Datenstruktur enthält alle *message events*, deren Auftreten zu einem Zustandswechsel führt. Jedem *message event* ist ein *modal message event* zugeordnet. Die Klasse *ModalMessageEvent* wurde um Listen von aktiven Diagrammen erweitert, in denen das Event *enabled, violating, etc.* ist. Die Modalität des MessageEvents enthält zusätzlich die Atttribute *hot, cold violating, cold, monitored* und *initializing*. Das letzte Attribut wird für Nachrichten gesetzt, die zur Aktivierung von Diagrammen führen.

AKTIVE PROZESSE In SCENARIOTOOLSIST es möglich, neben MSDs Zustandsautomaten, sogenannte Modal State Structures (MSSs) in die Simulation mit einzubeziehen. Das Konzept der aktiven Kopie wird dafür verallgemeinert auf einen aktiven *Prozess* . Jeder aktive Prozess enthält MessageEvents, die abhängig vom Zustand des Prozesses *violating* oder *enabled* sind. Zusätzlich ist es möglich, auf den Wert gebundener Variablen zuzugreifen (*VariableValuations*).

ActiveProcess

OBJEKTSYSTEM Enthält die jeweils nach einem Simulationsschritt aktualisierte Version des Instanzenmodells. Zusätzlich werden alle *message events* und *runtime messages* in einem *MessageEventContainer* gespeichert.

3.2.3 *Laufzeitverhalten*

Zu Beginn der Simulation wird ein *runtime state graph* mit einem einzigen Zustand erstellt. Dieser enthält die Objekte aus dem geladenen Instanzenmodell sowie alle Nachrichten, die von Environment-Objekten zu System-Objekten gesendet werden und zur Aktivierung

eines *Requirement-MSD* führen. Diese Situation entspricht der Annahme, das das System zu Beginn inaktiv ist und auf Nachrichten des Environments wartet. Die Liste der initialisierenden Nachrichten wird zu Beginn der Simulation aus der UML-Spezifikation extrahiert. Dazu wird die Klasse *RuntimeUtil* genutzt, die eine große Anzahl an UML-spezifischen Funktionen enthält.

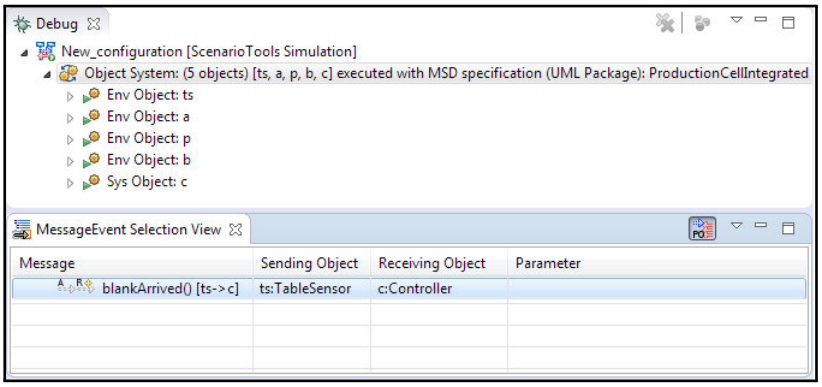

Abbildung 3.6: Simulations-Perspektive von ScenarioToolsam Anfang einer Simulation

Startet man die Simulation der ProductionCell, kommt zu Beginn nur die Nachricht $\texttt{blankArrived}\ (\texttt{ts} \rightarrow \texttt{c})$ in Frage (siehe Abbildung 3.6)), da alle anderen Nachrichten zu keinem Zustandswechsel führen.

hier blankArrived!

Um einen Simulationsschritt durchzuführen, muss die *performStep*-Methode des aktuellen Zustands mit einem Event aus der *event-map* aufgerufen werden (Siehe Algorithmus 3.1). Zunächst werden eventuelle Nebeneffekte des Events auf das Instanzenmodell ausgewertet (Z.1). Diese können auftreten, wenn das Event eine set*Property*-Methode repräsentiert und einen konkreten Parameter enthält. Anschließend wird der *cut* von jedem aktiven Diagramm hinter das Event verschoben, sofern es in diesem Diagramm vorkommt und *enabled* ist. Falls das Event nicht *enabled* ist, wird das aktive Diagramm verworfen. Wenn eine *safety violation* aufgetreten ist, wird diese im Zustand vermerkt. Wenn der Cut verschoben werden konnte, ist es möglich, das Invarianten und Zuweisungen ausgewertet werden können, dabei handelt es sich um sogenannte *hidden events*. Falls das MSD symbolische Lifelines enthält, wird geprüft, ob diese gebunden werden können. Die letzten beiden Schritte müssen gegebenfalls mehrmals hintereinander ausgeführt werden, da Invarianten und Zuweisungen von noch nicht gebundenen Lifelines abhängen und umgekehrt (Z.5-6). Anschließend wird für jedes MSD das mit dem gewählten Event beginnt, eine neue aktive Kopie erstellt (Z.7).

Im ersten Simulationsschritt sind tatsächlich nur die erste und die letzte Zeile relevant, da zu Beginn der Simulation keine Szenarien aktiv sind. Nachdem *blankArrived* ausgewählt wurde, wird eine neue aktive Kopie von *ArmATransportBlankToPress* erstellt, deren Cut sich hin-

Algorithmus 3.1: performStep(Event e)

```
1 performSideEffects(e) ;
2 foreach ActiveProcess p do
3     progressCut(p, e) ;
4     while p changed do
5         p.updateHiddenEvents(e);
6         p.updateLifelineBindings(e);
7 createActiveMSDs(e);
```

ter *blankArrived* und vor *pickUp* befindet (Cut Nr. 1 in Abbildung 3.7). Diese kapselt die UML-Interaktion, die von dem Sequenzdiagramm dargestellt wird. Der Cut der aktiven Kopie zeigt für jede Lifeline auf das *InteractionFragment*, das auf dieser Lifeline *enabled* ist. Während

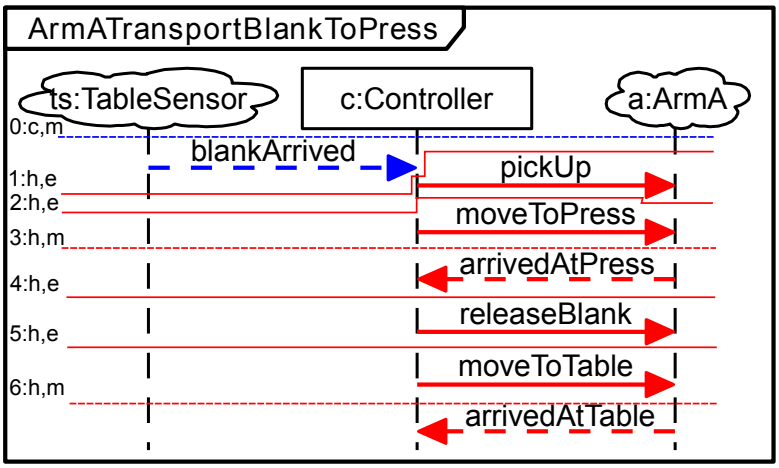

Abbildung 3.7: Alle möglichen *cuts* eines Szenarios. Durchgezogene *cuts* sind executed, blaue *cuts* cold, usw.

eine aktive Kopie erstellt wird, müssen alle Lifelines des Diagramms gebunden werden. Bei statischen Bindungen ist dies ohne weiteres möglich, da diese zu Beginn der Simulation definiert wurden. Für eine symbolische Lifeline müssen die Objekte bestimmt werden, die an diese gebunden werden dürfen. Dazu wird zunächst der zugehörige OCL-Ausdruck ausgewertet. Stimmt die Anzahl der gefundenen Objekte nicht mit der Multiplizität der Lifeline überein, wird die Lifeline „ignoriert“oder eine Fehlermeldung ausgegeben. Falls ein Objekt durch den OCL-Ausdruck bestimmt wurde, das nicht an die Lifeline gebunden darf, wird ebenfalls ein Fehler ausgegeben.

role2object mapping

vgl. UML2Ecore-Mapping

Der für den Playout-Algorithmus zentrale Teil der Implementierung erfolgt bei der (Neu-)Berechnung der Event-Map. Diese muss, sobald neue aktive Diagramme erstellt werden, oder der Cut von bestehenden Diagrammen verschoben wird, neu berechnet werden. Es werden zunächst alle Nachrichten betrachtet, die sich in aktiven Diagrammen befinden. Um das Playout-Verhalten umzusetzen, wer-

den aus diesen, abhängig davon ob gerade System oder Environment aktiv ist, bestimmte Nachrichten herausgefiltert. Um zu bestimmen, ob das System aktiv ist, wird über alle aktiven Diagramme iteriert und überprüft, ob es eine Nachricht gibt, die von einem Systemobjekt gesendet wird, *enabled* und *executed* ist. Ansonsten ist das Environment aktiv. Wenn das System aktiv ist, werden der Event-Map nur Nachrichten hinzugefügt, die von Systemobjekten gesendet werden und die in einem Requirement-MSD *executed* sind. Wenn das Environment aktiv ist, werden nur Nachrichten hinzugefügt, die Assumption-MSD *executed* sind. Bezeichne mit e ein MessageEvent, m ein ModalMessageEvent das e repräsentiert und n eine Nachricht in einem aktiven Diagramm, die von e repräsentiert wird. Dann wird die Modalität von m wie folgt bestimmt:

Message: Nachrichtendaten (Sender, Name, etc.)
MessageEvent: Nachrichtenart (synchron, symbolisch, etc.)
ModalMessageEvent: Modalitäten (hot, initializing etc.)

- ist n *enabled*, wird die Modalität von m mit der Modalität von n kombiniert.
- ist n *violating*, werden die *safety/cold violating*-Attribute der Modalität von m abhängig von der Modalität des Cuts aktualisiert.
- ist n *forbidden*, d.h. eine Nachricht die nicht auftreten darf, wird das *cold violating*- Attribut der Modalität von m aktualisiert wenn n cold forbidden ist, analoges gilt für Nachrichten die *hot forbidden* sind.
- ist e in der Liste der initialisierenden Events des *ObjectSystems*, wird das *initializing*-Attribut der Modalität von m aktualisiert.

BEISPIEL Nachdem *blankArrived* gesendet wurde, müssen die MessageEvents neu berechnet werden, da sich der Cut geändert hat. *pickUp* ist *enabled* (siehe Cut 1, Abbildung 3.7), daher wird die Modalität verwendet, die im Diagramm angegeben ist (*hot/executed*). Da der Cut hot ist, erhalten alle anderen Nachrichten die im Diagramm vorkommen die Modalität *safety violating*. In allen Fällen wird überprüft, ob bereits ein *ModalMessageEvent* existiert, das zum betrachteten *MessageEvent* gehört. Falls ja, werden die Modalitäten kombiniert. Zum Beispiel würde sich als Modalität von *arrivedAtTable* in der in Abbildung 2.5 dargestellten Situation ergeben, dass *arrivedAtTable hot/monitored* in einem *Requirements MSD* und *hot/executed* in einem *Assumption MSD* ist.

Für jedes parametrisierte konkrete Event wird parallel dazu auch ein symbolisches MessageEvent erstellt. Dieses entspricht dem konkreten MessageEvent bis auf die Tatsache, das es den Parameterwert nicht enthält. Sowohl abstrakte als auch konkrete Events werden durch die MessageEvent-Klasse modelliert. Jedes symbolische Event enthält eine Liste von konkreten Events, die zu diesem Event gehören. Umgekehrt hat jedes konkrete Event genau ein symbolisches *parent*-Event.

3.2.4 *Alternative Playout-Varianten*

Außer dem klassischen Playout können die folgenden beiden Ausführungsstrategien angewandt werden:

- *playout but waiting possible*
- *always all events*

Alle drei Varianten unterscheiden sich ausschließlich darin, welche Nachrichten für den nächsten Schritt zugelassen werden. Bei *playout but waiting possible* ist es möglich, das das System auf das Environment „warten kann“. Umgesetzt wird dieses Verhalten indem während eines Simulations-Schrittes, in dem das System an der Reihe ist, bestimmte Nachrichten des Environments der Event-Map hinzugefügt werden.

3.2.5 *Zustandsraumexploration*

Soll der Zustandsraum des Ausführungsverhaltens bestimmt werden, müssen die einzelnen Zustände im Stategraph gespeichert werden. Dieser bietet zwei Funktionen an, um einen, oder alle Nachfolgezustände eines gegebenen Zustandes zu berechnen.

Dabei ergibt sich das Problem, das bei der Durchführung eines Simulationsschrittes auf einem Zustand s_i der ursprüngliche Zustand verloren geht. Um diesen zu erhalten, wird stattdessen der komplette Zustand kopiert, und der Simulationsschritt auf der Kopie s_c ausgeführt. falls s_c zum ersten Mal besucht wurde, wird er im Stategraph gespeichert und es wird die Anzahl der gefundenen inkrementiert. Wenn es bereits einen Zustand s_o gibt, der äquivalent zu s_c ist, wird s_c verworfen. Dem Zustand s_i wird im ersten Fall eine Transition zu s_c, im zweiten Fall zu s_o hinzugefügt.

Bei der Simulation größerer Spezifikationen können sehr viele Zustände entstehen, sodass dementsprechend viel Speicher verbraucht wird. Um einen Teil des Speicherplatzes einzusparen, werden nach der Kopie und Aktualisierung eines Zustands Referenzen auf dessen Teile in einem Container abgelegt. Das Funktionsprinzip des Containers ist in Abbildung 3.8 illustriert. Falls der Zustand ein Objekt enthält (graue Kreise), zu dem kein äquivalentes Objekt im Container existiert (grüne Kreise), wird dieses dem Container hinzugefügt (oberer Teil der Abbildung). Wenn der Zustand Objekte enthält, zu denen äquivalente Objekte im Container existieren, werden diese durch die Objekte im Container ersetzt (siehe unterer Teil der Abbildung).

3.3 ERGEBNIS

Aufgrund der Struktur der vorhandenen Implementierung kann ein Großteil des Codes übernommen werden. Die meisten Abhängigkei-

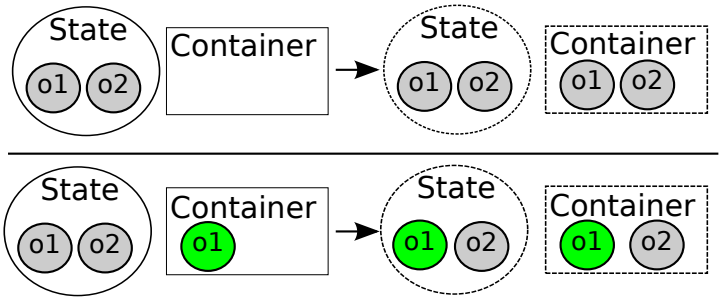

Abbildung 3.8: Caching-Prinzip. Oben: Füllen des Containers mit neuen Objekten. Unten: Ersetzen von Objekten im RuntimeState durch Objekte aus dem Container

ten zu UML liegen in den Klassen *MSDUtil*, *RuntimeUtil*, *ActiveMSD*, *ActiveMSS* und *MSDObjectSystem*. Da die Klasse *MSDUtil* nur zur Speicherung von UML-Diagrammnachrichten dient und die meisten Funktionen des *runtime util* durch den Einsatz von generiertem Code überflüssig werden, können beide Klassen entfernt werden. Die Klassen *ActiveMSD* und *ActiveMSS* sollten in der neuen Implementierung durch generierten Code ersetzt werden. Aus dem *MSDObjectSystem* müssen die UML-spezifischen Teile ersetzt oder entfernt werden.

4

RUNTIME-MODIFIKATION UND CODEGENERIERUNG

In diesem Kapitel werden die neue Runtime-Komponente und die Anforderungen an den generierten Code erläutert. Desweiteren wird exemplarisch gezeigt, wie Code aus modalen Sequenzdiagrammen generiert werden kann, der mit der neuen Runtime ausführbar ist.

4.1 ENTWURF UND IMPLEMENTIERUNG DER NEUEN RUNTIME-KOMPONENTE

Für den Entwurf wurde, ähnlich wie bei den vorhandenen Paketen von ScenarioTools ein Ecore-Modell verwendet. Das neue Modell ist strukturell ähnlich zum alten Modell. Es gibt jedoch einige Unterschiede. Die *role2eobject* und *uml2ecore* maps wurden aus dem Objektsystem entfernt, da die Runtime und der generierte Code unabhängig von UML sein soll. Desweiteren wurden die *Util*-Klassen entfernt, einige Helper-Funktionen jedoch übernommen. Schließlich wurde eine abstrakte Klasse *ActiveScenario* modelliert, die die Klassen *ActiveProcess* und *ActiveMSD* ersetzen soll. Diese enthält, ähnlich wie die Klasse *ActiveProcess* Mengen von *enabled, violating, hot forbidden, cold forbidden* events. *ActiveScenario* berechnet allerdings die Modalitäten dieser *events* in jedem Schritt selbst. Im Gegensatz zu *ActiveProcess* und *ActiveMSD* sind für die Klasse *ActiveScenario* jedoch kein *State* und keine *variable valuations* vorgegeben. Außerdem wurde die Referenz auf das Objektsystem entfernt. Diese wird stattdessen als Parameter übergeben um potenzielle Fehlerquellen einzuschränken. Um eine Helper-Methode zu sparen, wurde eine *enumeration ScenarioKind* eingeführt, die die zwei Literale *ENVIRONMENT_ASSUMPTION* und *SYSTEM_REQUIREMENT* enthält. Damit kann über die Methode *getKind()* direkt bestimmt werden, ob ein Szenario *assumptions* oder *requirements* darstellt. Zusätzlich wurden einige allgemeine Attribute modelliert, die während der Ausführung nützlich sein könnten:

- *currently hot/currently executed* wird jeweils true, wenn der aktuelle Cut *hot/monitored* ist,
- *violation occurred* wird true, wenn eine *safety / cold violation* auftritt,

- *finished* wird *true*, wenn alle Nachrichten im Szenario in der richtigen Reihenfolge eingetreten sind,
- *system / environment active* wird *true*, wenn das system oder das environment aktiv ist.

4.2 ANFORDERUNGEN AN DEN GENERIERTEN CODE

An den generierten Code werden folgende Anforderungen gestellt:

1. er ist korrekt
2. er implementiert die Schnittstelle von *ActiveScenario*
3. er baut auf dem EMF auf

1. ist wichtig, da automatisch generierter Code schwer zu verstehen und daher auch eventuell auch schwer zu debuggen sein kann. 1. und 2. sind wichtig, um ein korrektes Ergebnis bei der Ausführung der Spezifikation zu erreichen. 3. wird gefordert, um die aktiven Szenarien auch mittels EMF serialisieren zu können. Desweiteren sind durch Nutzung von EMF einige Probleme leichter lösbar. Zum Beispiel ist es durch die Nutzung einer EMF-Funktion möglich, die generierten Szenario-Klassen zu laden, wenn diese aus einem Ecore-Model generiert wurden.

Zur Codegenerierung wurde daher folgender Ansatz gewählt:

1. Schreibe Spezifikation in beliebiger Sprache
2. Generiere Ecore-Klassenmodell und Ecore-Spezifikations-Modell
3. Generiere Code aus beiden Modellen

Mit Klassenmodell ist hier das Modell gemeint, das die Klassen des modellierten Systems enthält. Das Spezifikationsmodell sollte eine Klasse für jedes Szenario enthalten. Im folgenden wird anhand des modalen Sequenzdiagrammes *ArmAMoveFromTableToPressAssumption* (im folgenden A1 abgekürzt), wie das Spezifikationsmodell und der Code für das Diagramm generiert werden kann.

4.2.1 *Ecore-Modell*

Für A1 wird dem Modell eine neue Klasse hinzugefügt, die von *ActiveScenario* erbt. Der Einfachheit halber sollte diese nach dem Namen des Szenarios, falls vorhanden, benannt werden. Anschließend wird der Klasse $\mathcal{A}1$ ein Attribut *state_A* sowie ein Ganzzahl-Attribut *state_C* hinzugefügt. Diese Attribute werden benötigt, um den Zustand/*cut* des MSDs zu speichern.

Desweiteren werden zwei Referenzen *c*, *a* vom Typ *Controller*, bzw. *ArmA* zu der Klasse hinzugefügt. Allgemein wird für jede Lifeline

eine Referenz mit deren Typ der Klasse hinzugefügt. Diese verweisen später auf konkrete Objekte die an diesem Szenario teilnehmen. Schließlich fügen wir zwei Referenzen vom Typ *ModalMessageEvent* hinzu und benennen sie mit *moveToPress* und *ArrivedAtPress*. Damit ist die Klasse $\mathcal{A}1$ komplett.

```
public class ArmAMoveFromTableToPress {
  Controller c; ArmA a;
  int stateC, stateA;
  ModalMessageEvent moveToPress;
}
```

Listing 1: Automatisch generierter Code (Ausschnitt)

Nachdem Code für die Klasse generiert wurde (z.B. mit einem Generator Model), muss die Klasse sechs Funktionen implementieren, die in *ActiveScenario* deklariert sind. Diese lauten wie folgt:

- *updateBindings(ObjectSystem)* Diese Methode versucht alle nicht initialisierten Referenzen an Objekte aus dem Objektsystem zu binden.
- *initializeMessageEvents(ObjectSystem)* Diese Funktion soll die modalen Nachrichten in dem Szenario erstellen, falls noch nicht geschehen.
- *updateMessageEvents(MessageEvent)* Diese Funktion aktualisiert die Listen *enabled, violating,...* events des Szenarios.
- *updateState(MessageEvent)* Diese Funktion führt zu einem Zustandswechsel des Szenarios.
- *updateHiddenEvents* Ein Aufruf dieser Funktion führt eventuell dazu, das *hidden events* aktualisiert werden.
- *updateReferences(Map<EObject,EObject>)* Diese Funktion ist notwendig, wenn sich das Instanzenmodell ändern kann. Dann müssen alle Objekte des Szenarios die in der Map sind ersetzt werden.

initializeMessageEvents und *updateBindings* müssen nach Erstellung des aktiven Szenarios und in jedem Simulationsschritt aufgerufen werden, in dem sich der Zustand des Szenarios ändert. *updateState* wird einmal pro Simulationsschritt aufgerufen, wenn das ausgewählte Event relevant für das Szenario ist. *updateHiddenEvents* kann, genauso wie *initializeMessageEvents* öfters als einmal pro Schritt aufgerufen werden. *updateReferences* muss dann aufgerufen werden, wenn dynamische Systeme simuliert werden, und sich der Zustand geändert hat.

4.2.2 *Implementierung von updateState*

Um den Zustandswechsel zu implementieren, wird der Ansatz von Greenyer benutzt[Gre10]. Die generelle Idee ist, für jede Lifeline zu zählen, wieviele Nachrichten schon von dieser Lifeline gesendet oder empfangen wurden. Die zuvor generierten Variablen *state_C, state_A* (Lifeline-Variablen) werden benutzt, um sich den Fortschritt auf jeder Lifeline zu merken. Dazu werden die Nachrichten auf jeder Lifeline von oben nach unten durchnummeriert. Eine Nachricht ist genau dann enabled, wenn die Lifeline-Variablen der sendenden und empfangenden Lifeline die gleichen Werte haben wie die Nachricht. Um den Zustand eines Szenarios zu aktualisieren, wird updateStep mit einer Fallunterscheidung für jede Nachricht gefüllt. Wenn die erste Nachricht nicht *enabled* ist, wird geprüft, ob die zweite Nachricht *enabled* ist, usw. Wenn das übergebene Event mit keiner Nachricht übereinstimmt, wird eine *violation ausgelöst*. Dieses Verhalten ist korrekt, wenn man vorraussetzt, das performStep nur mit Nachrichten aufgerufen wird, die auch relevant für das Szenario sind. Listing 2 zeigt einen Teil der generierten updateStep-Methode.

```
public void updateState(MessageEvent event) {
 if (isMoveToTableEnabled.eval() && isMessageUnifiable(event,
 moveToTable.getRepresentedMessageEvent())) {
 state_C++;state_A++;
 setCurrentlyExecuted(true);
 setCurrentlyHot(true);
 // ...
```

Listing 2: Fragment von updateStep für A1

für jedes *modal message event* überprüft, ob es *enabled* ist und ob das aufgetretene *event* damit unifizierbar ist. Ist dies der Fall, werden die Lifeline-Variablen der sendenden Lifeline um 1 erhöht. Abhängig von der Modalität der Folgenachrichten muss die Modalität des Szenarios geändert werden, dies geschieht durch die Aufrufe von *setHot* / *setExecuted*.

Modalität des Cut

4.3 PAR-FRAGMENTE

Für par-Fragmente muss für jeden Operanden des Fragments eine zusätzliche Variable für jede Lifeline generiert werden, der Grund hierfür ist, das ansonsten zwei oder mehr Positionen in einer Variable gespeichert werden müssten. Generell muss bei *CombinedFragments* darauf geachtet werden, das bedeckten Lifelines vor Eintritt am Rand des Fragments synchronisiert werden müssen. Die *enabled*-Bedingungen müssen entsprechend angepasst werden. Die letzten Nachrichten jedes Operanden müssen anstatt den Wert der Lifeline um 1 zu erhöhen, die Differenz zwischen der Position hinter dem Fragment und ih-

rer aktuellen Position dazuaddieren. Ansonsten wäre es möglich, das von einem Operanden in einen benachbarten Operanden gesprungen wird.

4.4 ALT-FRAGMENTE

Bei einem alt-Fragment gilt das gleiche wie bei einem par-Fragment, nur das keine zusätzlichen Variablen benötigt werden. Wenn keine guards angegeben sind, muss eine Zufallszahl den ausgewählten Operanden bestimmen. Wenn guards angegeben sind, müssen diese mit in die *isEnabled*-Funktion aufgenommen werden.

4.5 LADEN VON SZENARIEN

Zusätzlich zu den Szenario-Klassen wird eine Spezifikations-Klasse generiert, die Informationen darüber enthält, welche Szenarien von welchen events gestartet werden und welches Instanzenmodell benutzt werden soll.

4.6 BEDINGUNGEN

Bedingungen werden in der Methode *updateHiddenEvents* überprüft. Abhängig von der gewählten Semantik, kann unterschiedliches Verhalten für den Umgang mit Bedingungen festgelegt werden. Wenn eine Bedingung *enabled* ist, d.h. die Werte aller von der Bedingung überdeckten Lifeline-Variablen befinden sich vor der Bedingung, die Bedingung aber nicht ausgewertet werden kann, ist es entweder möglich zu warten, bis die Variable gebunden wird, oder eine violation auszulösen.

4.7 BINDINGS

Für die neue Runtime wurde der Binding-Mechanismus von den UML-Rollen abstrahiert. Um festzulegen, das ein bestimmtes Objekt in einem bestimmten Szenario mitwirken soll, kann das Objekt vor Beginn der Simulation mit einem Bezeichner im Objektsystem hinterlegt werden.

Dies könnte zum Beispiel in der Spezifikations-Klasse geschehen. Innerhalb der *updateBindings*-Funktion in einem konkreten Szenario kann dann über diesen Bezeichner das Objekt geholt werden. Für dynamische Bindings kann z.B. die Java Stream-API benutzt werden. Um zum Beispiel die Referenz b an irgendein Objekt zu binden, dessen X-Wert größer als 5 ist, kann der in Listing 3 verwendete Code benutzt werden. Für die Referenz a ist der Code für ein statisches Binding angegeben.

```
public void updateBindings(MessageEvent e, JObjectSystem os) {
 a = os.getStaticBindings.get("a");
 b = os.getControllableObjects().
     stream().filter(o -> o.getX > 5).getAny();
}
```

Listing 3: statische und dynamische Bindung von Objekten

4.8 PLAYOUT-STRATEGIEN

Um die runtime flexibler zu machen, wurde eine neue Klasse *ExecutionStrategy* definiert. In der generierten Spezifikation kann eine Referenz auf eine konkrete Strategie angegeben werden, um z.B. verschiedene Typen von Spezifikationen mit verschiedenen Playout-Strategien auszuführen. Um die Strategie-Klasse benutzen zu können, wurde die Sichtbarkeit einiger Methoden des *RuntimeState* geändert.

5

ZUSTANDSRAUMEXPLORATION

Um die Funktionalität der Runtime zu überprüfen, wurde ein zusätzliches Eclipse-Plugin geschrieben, das den Playout-Zustandsraum einer Spezifikation untersucht. Vom Startzustand aus werden alle erreichbaren Knoten in Tiefensuche durchlaufen. Falls keine neuen Zustände mehr gefunden werden, terminiert der Algorithmus. Der erzeugte Zustandsraum wird serialisiert und im XML-Format gespeichert. Da die Suche nach bestimmten Informationen in dieser Datei sehr aufwändig ist, sobald größere Spezifikationen simuliert werden, wurde zusätzlich ein Ausgabeprogramm geschrieben, das mittels dem Graph-Layouter *GraphViz*[1] ein Zustandsübergangsdiagramm im PDF-Format erzeugt. Für einen Teil der ProductionCell Spezifikation ist der erzeugte Zustandsgraph in Abbildung 5.1 angegeben. Wenn eine *safety violation* in einem *assumption scenario* aufgetreten ist, wird der Zustand rot markiert, wenn eine *safety violation* in einem *requirement scenario* aufgetreten ist, wird der Zustand blau markiert.

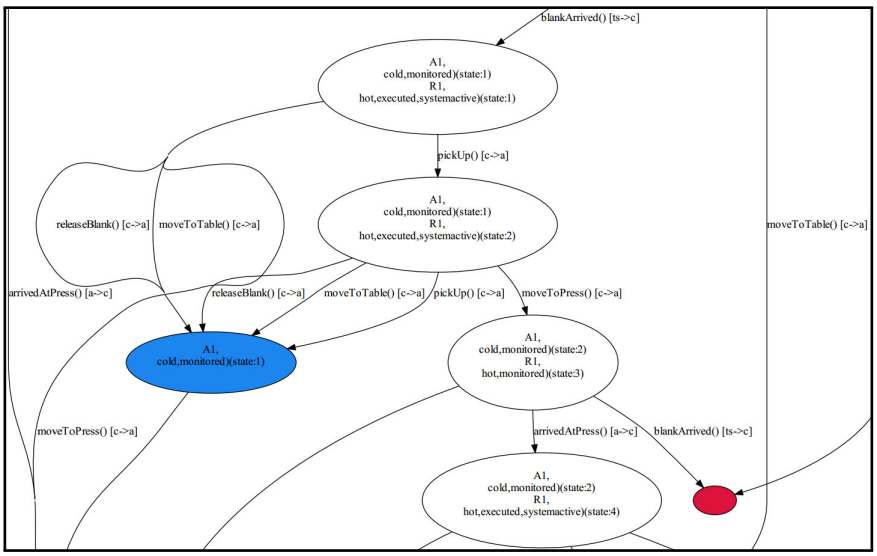

Abbildung 5.1: Ausschnitt aus dem Zustandsraum der ProductionCell. Der Übersicht halber wurden nur zwei Szenarien simuliert.

1 http://www.graphviz.org/

6

VERWANDTE ARBEITEN

In diesem Kapitel werden einige verwandte Arbeiten vorstellt.

Kundu et al. verwenden einen *Sequence Integration Graph* (SIG) um Java-Code aus Sequenzdiagrammen zu generieren, dabei steht allerdings nicht die Simulation bzw. der Test der Spezifikation im Vordergrund, sondern die Erzeugung von direkt nutzbarem Programmcode [KSM13]. Der generierte Programmcode ist zudem nicht immer vollständig, da nur Sequenzdiagramme betrachtet werden und diese z.B. keine Informationen über Klassen beinhalten.

Maoz und Harel haben den *S2A* Compiler vorgestellt [MH06]. S2A kann *Live Sequence Charts*, eine grafische Modellierungssprache mit ähnlichen Eigenschaften wie MSDs, in aspektorientierten Java-Code übersetzen. In einem aspektorientierten Programm überwachen *Aspekte* den Programmfluss. Es ist möglich, durch sogenannte pointcuts festzulegen, das an bestimmten Stellen im Programmcode (join points) zusätzlicher Code durch sogenannte advices ausgeführt wird. S2A generiert für jedes LSC einen Aspekt und zusätzlich einen *coordinator*, der die Ausführung überwacht. Ein Aspekt kann den Aufruf von Funktionen während der Ausführung verfolgen, und dabei automatisch seinen Cut aktualisieren. Der Unterschied zu dem in dieser Arbeit generierten Code besteht darin, das die Erweiterung von ScenarioTools keine Aspekte verwendet. Außerdem ist direkte Ausführung des generierten Codes in ScenarioToolsso wie bei S2A so nicht möglich. Der von S2A generierte *Coordinator* erfülllt eine ähnliche Rolle wie der *runtime state* in der ursprünglichen Version. Die Idee, das Playout-Verhalten in einer Strategie-Klasse zu verpacken wurde von S2A übernommen.

Derzeit wird eine domänenspezifische Sprache am Fachgebiet Software Engineering entwickelt, die in ScenarioToolsintegriert werden soll. Es wird daran gearbeitet, diese Sprache in den in dieser Arbeit vorgestellten Code zu übersetzen.

7

ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurde eine Methode vorgestellt, um Code aus einfachen modalen Sequenzdiagrammen zu generieren. Die Software ScenarioTools wurde um die Möglichkeit erweitert, diese Code auszuführen. Durch die Erweiterung der Software ist es jetzt prinzipiell möglich, Code für beliebige Szenarien auszuführen, solange diese auf Nachrichten bzw. Events basieren. Durch die Reduzierung der Schnittstelle auf erlaubte und verbotene Nachrichten, sowie der Auslagerung des PlayOut-Verhaltens in eine (erweiterbare) Strategie-Klasse ist die Software insgesamt flexibler geworden.

Als ein Nachteil der derzeitigen Implementierung hat sich die Performance herausgestellt. Die Gründe hierfür konnten leider nicht ermittelt werden, denkbar sind jedoch eine lange Ladezeit der kompilierten Java-Klassen, oder nicht optimierte Algorithmen beim Vergleich von Zuständen. Für die weitere Zukunft sollte untersucht werden, wie der generierte Code noch effizienter gestaltet werden kann. Außerdem könnte eine API entwickelt werden, die das Einbinden neuer Sprachen erleichtert. In diesem Zusammenhang wäre eine Einbindung von Tools wie JET oder Acceleo[1] interessant.

1 Programme zum Erstellen von Dokumenten aus erweiterbaren Templates

LITERATURVERZEICHNIS

[AS87] Bowen Alpern and FredB. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987. (Cited on page 5.)

[BGP13] Christian Brenner, Joel Greenyer, and Valerio Panzica La Manna. The scenariotools play-out of modal sequence diagram specifications with environment assumptions. In *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, volume 58. EASST, 2013. (Cited on pages 6 und 7.)

[GBC+13] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE 13)*, ESEC/FSE 2013, pages 433–443, New York, NY, USA, 2013. ACM. (Cited on page 7.)

[Gre10] Joel Greenyer. Synthesizing Modal Sequence Diagram specifications with Uppaal-Tiga. Technical Report tr-ri-10-310, University of Paderborn, February 2010. (Cited on page 24.)

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. (Cited on page 7.)

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008. (Cited on page 5.)

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Berlin Heidelberg, 1985. (Cited on page 3.)

[KSM13] D. Kundu, D. Samanta, and R. Mall. Automatic code generation from unified modelling language sequence diagrams. *Software, IET*, 7(1):12–28, February 2013. (Cited on page 29.)

[MH06] Shahar Maoz and David Harel. From multi-modal scenarios to code: Compiling lscs into aspectj. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 219–230, New York, NY, USA, 2006. ACM. (Cited on page 29.)

[OMG11] OMG. Omg unified modeling language (omg uml), superstructure, version 2.4.1, August 2011. (Cited on page 3.)

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 1.)

ERKLÄRUNG DER SELBSTÄNDIGKEIT

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, 16.September 2014

Nils Glade

INHALT DER BEILIEGENDEN CD

- Snapshot des ScenarioTools-Repositories vom 16.09.2014.
- Eine Kopie dieser Arbeit

Der für diese Arbeit erstellte Quellcode befindet sich im Verzeichnis scenariotools/org.scenariotools.java/plugins.
Beispielcode für das ProductionCell-Beispiel befindet sich im Verzeichnis scenariotools/org.scenariotools.java/examples/ProductioncellTest.
Starte Zustandsraumexploration über Rechtsklick auf generierte SpecImpl.java