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

On-the-fly Synthese für szenariobasierte Spezifikationen von Produktlinien

Bachelorarbeit

im Studiengang Informatik

von

Wasja Brunotte

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

Hannover, 28.03.2015

Erklärung der Selbstständigkeit

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

Hannover, den 28.03.2015

Wasja Brunotte

Zusammenfassung

Heutzutage muss Software für Autos, Industrieanlagen oder auch in der Medizin komplexe und sicherheitskritische Aufgaben übernehmen. Zudem handelt es sich bei Software für z.B. Autos, Flugzeuge, etc. um reaktive Systeme. Das heißt, sie stehen in ständiger Interaktion mit der Umwelt.

Die Software für solche Systeme zu entwickeln, ist eine große Herausforderung. Häufig müssen ihre Komponenten mehrere Funktionen gleichzeitig erfüllen, was ein präzises, systematisches Vorgehen von Seiten der Entwickler erfordert, die diese Abläufe für diese Systeme planen und umsetzen.

Um solche Abläufe präzise zu beschreiben, werden im formalen szenariobasierten Entwurf Modal Sequence Diagrams (MSDs) eingesetzt. So kann ein einzelnes Szenario beschreiben, was die Komponenten des Systems in bestimmten Situationen tun können, tun müssen oder nicht tun dürfen. Noch komplexer wird der Entwurf solcher Systeme, wenn sogenannte Produktlinien entworfen werden sollen. Produktlinien sind unterschiedliche Varianten eines Produkts, die verschiedene Kombinationen von Funktionalitäten vereinigen. Damit kann gezielter auf individuelle Kundenwünsche eingegangen werden, wie z.B. Ausstattungen bei Autos.

Mit der Werkzeugumgebung ScenarioTools kann der Entwickler nun Szenarien durch MSDs modellieren, anschließend ausführen und simulieren. Innerhalb ScenarioTools kann dann geprüft werden, ob eine Spezifikation konsistent ist und ob es realisierbare Produkte einer Produktlinie gibt. Hierzu soll in dieser Arbeit der Synthese-Algorithmus von Gressi [10] optimiert werden.

Abstract

Nowadays software is part of many safety critical systems and is needed to perform complex tasks. Software for cars, planes and so on are reactive systems. Reactive systems keep an ongoing relationship with its environment. They often consist of a set of components that provide various functions by their interaction and these interactions often have to satisfy complex specifications. Therefore engineers need intuitive, yet precise way for specifying the requirements for those systems.

To deal with these challenges Modal Sequence Diagrams (MSDs) are used in the formal scenario-based design. They allow engineers to specify sequences of interactions that may, must, or must not happen in a system.

Furthermore often engineers develop several variants of a system, a so called product line. The different products of a product line consist of several functions or components to fulfill individual user needs, e.g. cars.

ScenarioTools is an Eclipse-based tools suite which supports the engineer in modeling of MSDs. In addition MSDs can be executed and simulated. In this thesis the featured synthesis from Gressi [10] should be optimized by making these technique on-the-fly. The synthesis algorithm checks whether a specification is consistent and derives the possible combination of realizable products for a product line specification.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage sind Software-gesteuerte Systeme allgegenwärtig wie zum Beispiel in Autos, Industrieanlagen oder auch in der Medizin. Solche Software-intensiven Systemen, in denen Software das System grundlegend in Design, Konstruktion, Einsatz und Evolution beeinflusst, [17] müssen zudem häufig komplexe und sicherheitskritische Aufgaben übernehmen. Diese Systeme bestehen häufig aus mehreren Komponenten, die miteinander interagieren und auch gleichzeitig Aufgaben ausführen. Die Komponenten reagieren hierbei selbstständig auf Ereignisse der Umgebung in der sie laufen und stimmen sich mit dieser ab. Daher werden diese Systeme *reaktive Systeme* genannt.

Als Beispiel seien hier autonom fahrende Roboter in der Industrie angeführt, die als Transportmittel für schwere Güter genutzt werden oder Abstandsregelsysteme in Kombination mit Tempomaten bei PKWs. Hier soll das System zum einen die eingestellte Geschwindigkeit halten, muss zum anderen aber den vorausfahrenden Verkehr kontrollieren und das Fahrzeug ggf. abbremsen oder sogar anhalten.

Die Software für solche Systeme zu entwickeln, ist eine große Herausforderung. Häufig müssen ihre Komponenten mehrere Funktionen gleichzeitig erfüllen, was ein präzises, systematisches Vorgehen von Seiten der Entwickler erfordert. Anfangs steht meist die Spezifikation von Use Cases. Jedoch können beim szenariobasierten Entwurf Widersprüche entstehen. Es kann z.B. in einem Szenario etwas gefordert werden, was in einem anderen Szenario verboten ist. Beispielsweise könnte das Abstandsregelsystem aus obigem Beispiel dahingehend erweitert werden, dass ein Kamerasystem die zulässige Höchstgeschwindigkeit aus den Verkehrszeichen liest und diese dem Tempomaten mitteilt. Ist nun das Fahren mit einer höheren Geschwindigkeit erlaubt, würde der Tempomat den Beschleunigungsvorgang einleiten. Sollte allerdings ein vorausfahrendes Fahrzeug langsamer sein besteht hier ein Widerspruch. Der Abstandsregler meldet „bremsen“, wohingegen der Tempomat „beschleunigen“ meldet. Hier liegt nun ein Widerspruch vor, da

die Szenarien unterschiedliche Aktionen (Abbremsen und Beschleunigen) fordern. Demzufolge ist die Spezifikation nicht realisierbar.

Um solche Abläufe präziser zu beschreiben, werden im formalen szenariobasiertem Entwurf *Modal Sequence Diagrams* eingesetzt. So kann ein einzelnes Szenario beschreiben, was die Komponenten des Systems in bestimmten Situationen tun können, tun müssen oder nicht tun dürfen.

1.1 Problemstellung

Eine sich zur Zeit in der Forschung befindliche Werkzeugumgebung für den szenariobasierten Entwurf ist ScenarioTools. Mit deren Hilfe kann der Entwickler Szenarien durch *Modal Sequence Diagrams* modellieren, anschließend ausführen und simulieren. Somit kann geprüft werden, ob eine Spezifikation konsistent bzw. realisierbar ist. Also keine Widersprüche enthält. Noch komplexer wird der Entwurf solcher Systeme, wenn sogenannte *Produktlinien* entworfen werden sollen. Produktlinien sind unterschiedliche Varianten eines Software-Produkts, die verschiedene Kombinationen von Funktionalitäten vereinigen. Damit kann gezielter auf individuelle Kundenwünsche eingegangen werden, wie z.B. bei Autos.

„Je mehr Funktionalitäten der Produktlinie hinzugefügt werden, kann die Anzahl verschiedenen Varianten exponentiell wachsen. Hier nun sicherzustellen, dass jedes Produkt seine Spezifikation erfüllt, ist eine große Herausforderung im Produktlinien-Engineering.“, so Greenyer et al. [8].

1.2 Lösungsansatz

Die verschiedenen Funktionalitäten einer Produktlinie können in ScenarioTools mit Hilfe von *Feature Diagrams* realisiert werden. Das Verhalten der Funktionalitäten wird durch *Modal Sequence Diagrams* modelliert. Die Prüfung einer solchen Produktlinienspezifikation auf Widersprüche erfolgt dann durch einen Synthesealgorithmus wie in Abbildung 1.1 dargestellt.

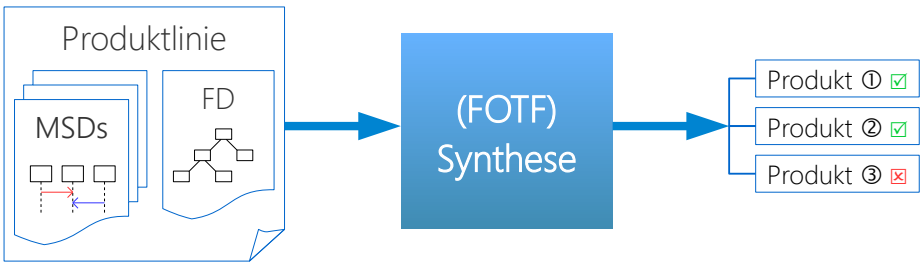


Abbildung 1.1: On-the-fly Synthese für Produktlinien

Ein Ansatz für eine *featured Synthese* für Produktlinien wurde bereits von

Gressi [10] entwickelt. Diese vorliegende Arbeit beschäftigt sich nun mit einer möglichen Optimierung des auf Gressi basierenden Algorithmus. Ziel hierbei ist die Geschwindigkeit der Synthese zu verbessern, indem der Algorithmus *on-the-fly* arbeitet. Das heißt das immer nur Teile der intern verwendeten Datenstruktur untersucht werden.

1.3 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert. Kapitel 1 gibt eine kurze Einleitung in das Thema, erläutert den Lösungsansatz und beschreibt die Struktur der Arbeit.

In Kapitel 2 werden die technischen Grundlagen für diese Arbeit vermittelt. Dazu zählen die Definition und Erläuterung von Begriffen und der verwendeten Technologien wie zum Beispiel *Produktlinien*, *Modal Sequence Diagrams* und *Feature Diagrams*. Zudem wird ein Überblick über das verwendete Werkzeug *Scenario Tools* gegeben.

Kapitel 3 beschreibt den erarbeiteten Synthesealgorithmus, der eine mögliche Lösung des eingangs erwähnten Problems darstellen soll.

Kapitel 4 beschäftigt sich mit der Implementation, die aus zwei Eclipse Plug-Ins besteht. Des weiteren wird die generelle Architektur dieser Plug-Ins beschrieben so wie deren Anwendung. Im Kapitel 5 erfolgt eine Zusammenfassung dieser Arbeit und es wird ein Fazit gezogen.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die grundlegenden Konzepte, Verfahren und Technologien vor, die in dieser Arbeit genutzt werden. Abschnitt 2.1 erläutert das Konzept der Produktlinien und erklärt, was darunter zu verstehen ist. Anschließend wird in Abschnitt 2.2 ein Beispiel eingeführt, das im weiteren Verlauf der Arbeit immer wieder aufgegriffen wird, um neue Konzepte zu veranschaulichen.

Abschnitt 2.3 beschäftigt sich mit Modal Sequence Diagrams (MSDs) und wie reaktive Systeme damit beschrieben werden können. In 2.4 werden Feature Diagrams (FDs) erläutert und dargestellt, wie sich das Verhalten von Produktlinien beschreiben lässt. Der Abschnitt 2.5 stellt ein algorithmisches Verfahren zum Ausführen von Modal Sequence Diagrams vor. Danach werden die in dieser Arbeit zur Anwendung kommenden Featured Game-Graphen in 2.6 diskutiert. In Abschnitt 2.7 wird ScenarioTools vorgestellt, eine Eclipse basierte Entwicklungsumgebung, mit der Produktlinien modelliert werden können und worin der Algorithmus dieser Arbeit implementiert ist. Im letzten Teil 2.8 werden verwandte Arbeiten vorgestellt und es wird ein kurzer Überblick über den aktuellen Stand der Technik in Bezug auf den Algorithmus gegeben.
Zur Einführung in die Produktlinien folgt nun ein kurzer historischer Abriss über die Großserienfertigung.

2.1 Produktlinien

Nach Gressi [10] und Alizon et al. [1] wird die Großserienfertigung Henry Ford zugeschrieben. Er hatte sich überlegt, welche Teile in der Autofertigung Wiederverwendung finden könnten in unterschiedlichen Automodellen. Daraus resultierte die Modell T Plattform, so dass schließlich mit dem gleichen Unterbau verschiedene Modell T Modelle realisiert werden konnten. Somit konnte die Produktion teilweise ausgelagert werden an darauf spezialisierte Firmen. Das war der Weg weg von kleinen Mengen handgefertigter Güter hin

zur Produktion von standardisierten Gütern in großen Mengen. Das Modell T war eines der erfolgreichsten Produkte der Automobilgeschichte. Das ist darauf zurückzuführen, dass die Produktionskosten durch die gemeinsamen Komponenten für verschiedene Modelle gesenkt werden konnten, was wiederum zu einem günstigeren Produkt für den Automarkt führte. Ein weiterer gewichtiger Vorteil war, die Möglichkeit der hohen Individualisierbarkeit in Hinblick auf die Bedürfnisse des Kunden.

Dieses Grundkonzept der Produktlinien ist bis heute in verschiedenen Bereichen der Wirtschaftsgüter wiederzufinden, wie zum Beispiel bei nahezu jedem Automobilhersteller. Der Kunde kann ein Produkt aus einer *Basis*-Produktlinie auswählen und diese schließlich individuell an seine eigenen Bedürfnisse anpassen. Weitere Beispiele für Firmen, die Produktlinien einsetzen sind Airbus, Dell und McDonald's [15].

Definiert ist der Begriff der Produktlinie (PL) von Clements et al. [3] wie folgt. Eine Produktlinie ist eine Gruppe von Produkten, die eine gemeinsam geführte Menge an Funktionalitäten teilen, um damit die Bedürfnisse eines bestimmten Marktsegments oder Auftrages bzw. Einsatzes zu befriedigen.

Im Software Engineering existieren ebenfalls Produktlinien. Die Software Produktlinie (SPL). Bei einer SPL wird die Produktlinie durch eine Menge von Software-Anwendungen repräsentiert. Clements et al. [16] definieren eine SPL als eine Menge software-intensiver Systeme, die eine gemeinsam geführte Menge an Funktionalitäten teilen, um damit die Bedürfnisse eines bestimmten Marktsegments oder Auftrages bzw. Einsatzes zu befriedigen und die von einer gemeinsamen Menge von Grundanforderungen entwickelt wurden in einer vorgeschriebenen Art und Weise.

Im Rahmen dieser Arbeit soll hier nicht weiter auf Software Produktlinien eingegangen werden, da sie ein ähnliches Konzept verfolgen und sich diese Arbeit mit dem allgemeinerem Konzept der Produktlinien beschäftigt. Im Folgenden Kapitel 2.2 wird ein Beispiel eingeführt, dass eine Produktlinie veranschaulichen soll.

2.2 Einführung des Beispiels

Das in dieser Arbeit vorliegende Beispiel behandelt einen Geldautomaten wie er in Bankfilialen, Einkaufszentren, Tankstellen oder anderen öffentlichen Gebäuden zum Einsatz kommt. Es soll dem besseren Verständnis der Grundlagen dienen und diese an den entsprechenden Stellen exemplarisch veranschaulichen.

Zur Vereinfachung werden sicherheitskritische Aspekte wie das Abfragen einer PIN oder das Einbehalten der EC-Karte nach mehrmaligem falschen Eingeben der PIN außer Acht gelassen. Abbildung 2.1 zeigt schematisch den Geldautomaten, der in diesem Beispiel behandelt wird.
Der Geldautomat nimmt in diesem Beispiel eine EC-Karte des Benutzers

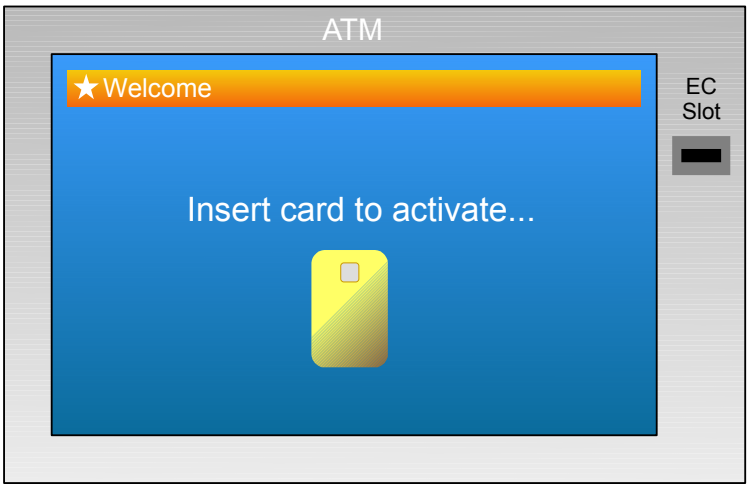


Abbildung 2.1: Beispiel Geldautomat mit Touchscreen-Eingabe

entgegen und bietet zwei verschiedene Geldfunktionen. Zum einen „Bargeld auszahlen“ und zum anderen „Geldkarte aufladen“. Mit Geldkarte ist hier eine nicht näher beschriebene Funktion der EC-Karte gemeint, mit der man bargeldlos bezahlen kann wie zum Beispiel an einem Fahrkartenautomaten. Folgende Entitäten sind an diesem Szenario beteiligt:

- Benutzer, der die EC-Karte in den Geldautomaten steckt, um ihn zu aktivieren und am Ende die EC-Karte ggf. mit dem Bargeld entgegen nimmt.
- Dispenser, der sich um die Auszahlung des Geldes und die Ausgabe der EC-Karte kümmert.
- Controller, der das Gesamtsystem steuert und mit dem Dispenser interagiert, damit dieser Bargeld und EC-Karte ausgibt.

Die Anforderungen sind im Folgenden informal beschrieben:

- **R1** Der Geldautomat wird durch Einschieben einer EC-Karte aktiviert. Es erfolgt die Aufforderung an den Benutzer die Geldfunktion zu wählen. Zudem kann die optionale Funktion „Sprachausgabe“ ausgewählt werden.
- **R2** Nachdem die Geldfunktion gewählt wurde, erfolgt die Aufforderung an den Benutzer den Geldbetrag zu wählen, der ausgezahlt oder mit dem die Geldkarte aufgeladen werden soll.
- **R3** Bei Bargeldauszahlung soll erst die EC-Karte ausgegeben und entnommen werden, bevor das Bargeld ausgegeben wird.

Des weiteren wird folgende Annahme gemacht:

- **Annahme 1** Die Sprachausgabe fordert bei „Bargeld auszahlen“ erst zur Entnahme des Geldes aus. Anschließend soll die EC-Karte entnommen werden.

2.3 Modal Sequence Diagrams

2.3.1 Einführung

Modal Sequence Diagrams (MSDs) basieren auf Live Sequence Charts (LSCs) [11]. Ein graphischer Formalismus, der dazu dient, die Anforderungen bei der Interaktion von Systemkomponenten und deren Umwelt zu definieren [10]. Bei MSDs kann zwischen *existential MSDs* und *universal MSDs* unterschieden werden. *Existential MSDs* beschreiben einen Ablauf an Ereignissen, die in mindestens einem Durchlauf möglich sein müssen für ein Objektsystem, so Greeyner [7]. *Universal MSDs* dagegen, so Greenyer weiter, formulieren Anforderungen, die immer gelten müssen bei jedem Durchlauf. In dieser Arbeit werden nur *universal MSDs* betrachtet und im Detail beschrieben. Hierzu wird als erstes etwas zum syntaktischen Aufbau von MSDs 2.3.2 erklärt. Anschließend der Ablauf und die Semantik 2.3.3 erläutert. Danach werden MSD Violations 2.3.4 beschrieben. Im Anschluss daran Assumption MSDs und MSD Spezifikationen 2.3.5. Schließlich noch Parametrisierung und verbotene Nachrichten im letzten Abschnitt 2.3.6.

2.3.2 Aufbau (Syntax)

Wie in 2.3.1 erwähnt sind MSDs ein grafischer Formalismus. Zur Veranschaulichung ist in Abbildung 2.2 ein MSD dargestellt.

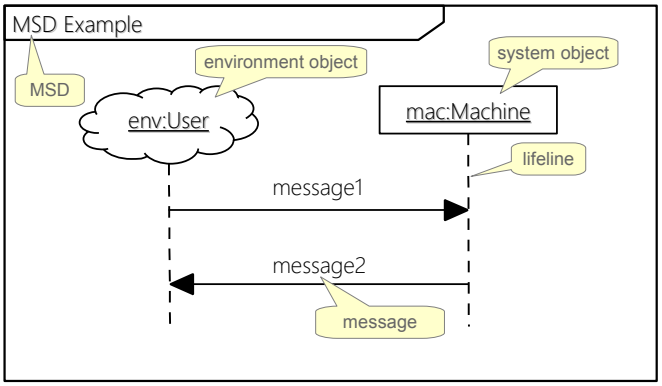


Abbildung 2.2: Vereinfachtes Beispiel eines MSDs

Es handelt sich hierbei um universelles MSD, was durch den durchgehenden Rahmen zum Ausdruck kommt. Links oben ist der Name des MSD zu lesen. Hier `MSD Example`. Innerhalb sind zwei Objekte zu sehen, sie gehören zum Objektsystem. Es wird zwischen System- und Umweltobjekten unterschieden. Letztere sind durch eine Wolke dargestellt. Systemobjekte werden mit einem Rechteck gezeichnet. Objekte erhalten einen Namen und einen Typ. Hier `mac:Machine` für das Systemobjekt und `env:User` für das Umweltobjekt. Jedes Objekt besitzt eine Lebenslinie, die gestrichelt dargestellt wird und als vertikaler Zeitstrahl angesehen werden kann (siehe 2.3.3). Objekte, sowohl System- wie auch Umweltobjekte, können Nachrichten untereinander austauschen. Sogenannte *MSD messages* oder auch *Diagramm messages*. Hierbei sendet ein Objekt, der Sender, eine message an ein anderes Objekt, den Empfänger. Dies wird durch Linien mit Pfeilen zwischen den Lebenslinien dargestellt. Hierbei startet die Linie beim Sender und endet mit Pfeilspitze beim Empfänger. Die MSD messages sind mit einem eindeutigen Namen versehen und können parametrisiert werden (siehe 2.3.6). Im Falle des Beispiels in Abbildung 2.2 ist `env:User` Sender der Nachricht `message1` und `mac:Machine` deren Empfänger. Der Vorgang des Sendens und Empfangens wird als *message event* oder nur *event* bezeichnet. Im Rahmen dieser Arbeit werden nur synchrone Nachrichten betrachtet, also Nachrichten, bei denen das Senden und Empfangen als ein *event* aufgefasst werden kann.

Mit MSDs können zudem Nachrichten definiert werden, die innerhalb des Systems auftreten müssen, auftreten können oder die nicht auftreten dürfen. Ausgedrückt wird das, indem man den Nachrichten zwei Modalitäten vergibt: die *temperature* und die *execution kind*. Abbildung 2.3 stellt diese Modalitäten der vier Kombinationsmöglichkeiten dar.

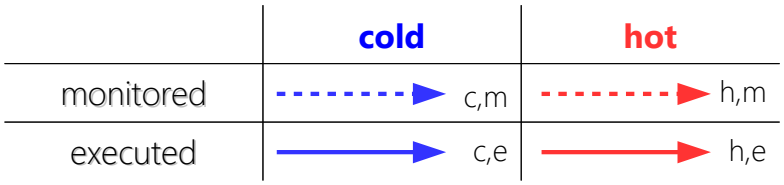


Abbildung 2.3: Mögliche Kombinationen von temperature und execution kind einer MSD message. In Anlehnung an [9].

Die *temperature* kann entweder cold oder hot sein. Ausgedrückt durch einen blauen Pfeil für cold und hot wird mit einem roten Pfeil ausgedrückt. Die *execution kind* kann entweder *executed* (e) oder *monitored* (m) sein. Hierbei bedeutet *executed* (durchgezogene Linie), dass diese Nachricht auftreten muss, wohin gegen *monitored* (gestrichelte Linie) bedeutet, dass eine Nachricht auftreten kann, aber nicht muss. Abbildung 2.4 zeigt ein MSD für das Geldautomatenbeispiel, dessen Nachrichten Modalitäten besitzen. Als erstes wartet das System `mac:Machine` darauf, dass der Benutzer seine EC-Karte

einsteckt. Dieses Ereignis kann eintreten, muss aber nicht. Es ist schließlich cold und monitored. Anschließend, nachdem der Benutzer seine EC-Karte eingesteckt hat, sendet das System dem Dispenser (`dis:Dispenser`) die hot, executed Nachricht `lockSlot`. Daraufhin muss der Dispenser den Karten-Slot verriegeln, damit z.B. keine weitere EC-Karte eingesteckt werden kann. Danach erwartet das System wieder eine Eingabe von Seiten des Benutzers (`chooseFunction`). Zum Schluss fordert das System den Dispenser auf, die EC-Karte des Benutzers auf jeden Fall auszugeben. Es sei an dieser Stelle darauf hingewiesen, dass dieses Beispiel der Veranschaulichung dient, was MSD Nachrichten mit Modalitäten angeht. Sicherlich müsste der Dispenser vor dem Ausgeben der EC-Karte noch eine `unlockSlot` Nachricht erhalten und es würde auch eine komplexere Interaktion zwischen Umwelt (Benutzer) und System stattfinden.

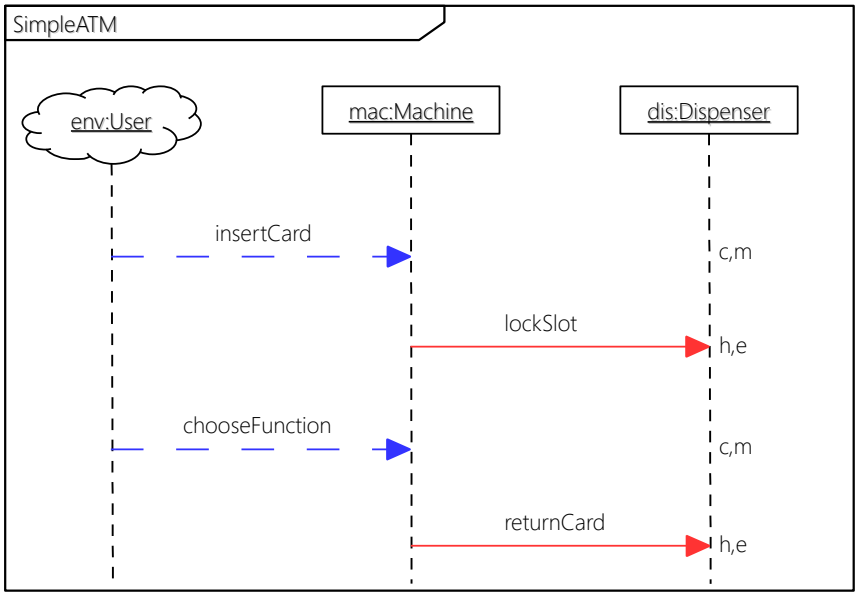


Abbildung 2.4: Beispielhaftes MSD für Geldautomaten, dessen Nachrichten Modalitäten tragen.

Im folgenden Abschnitt 2.3.3 wird die Semantik von MSDs und deren Ablauf diskutiert. Dieses wird anschließend wieder durch ein kurzes Beispiel veranschaulicht.

2.3.3 MSD Ablauf (Semantik)

MSDs werden sequentiell ausgeführt, d.h. die oberste Nachricht einer Lebenslinie stellt das *message event* dar. Beim Auftreten dieser ersten Nachricht wird eine neue Instanz des MSDs erzeugt, auch *active MSD* genannt. Jedes MSD kann während der Laufzeit nur ein *acitve MSD* haben.

Anschließend erwartet das active MSD dann eine oder mehrere weitere Nachrichten, wobei diese als *enabled* bezeichnet werden. Dieser Zustand, in dem sich das active MSD befindet, wird als *cut* bezeichnet. Dieser *cut* wird durch eine gestrichelte horizontale Linie dargestellt und besitzt genau wie die Nachricht selber auch Modalitäten. Hierbei erbt der cut die Modalitäten der nachfolgenden Nachricht. Ist die nachfolgende Nachricht cold, ist der cut ebenfalls cold. Ist die nachfolgende Nachricht hot, ist der cut ebenfalls hot. Um zu entscheiden, ob eine Nachricht auch die ist, die erwartet wird während des Ablaufs, muss diese *unifizierbar* sein. Eine Nachricht ist *unifizierbar* genau dann, wenn

1. die sendende und empfangene Lebenslinie der MSD Nachricht, die der sendenden und empfangenen Objekte im Objektsystem referenziert und

2. die MSD Nachricht und das Event im Objektsystem den gleichen Namen haben - also die gleichen Operationen sind und die evtl. Parameter vom gleichen Typ sind.

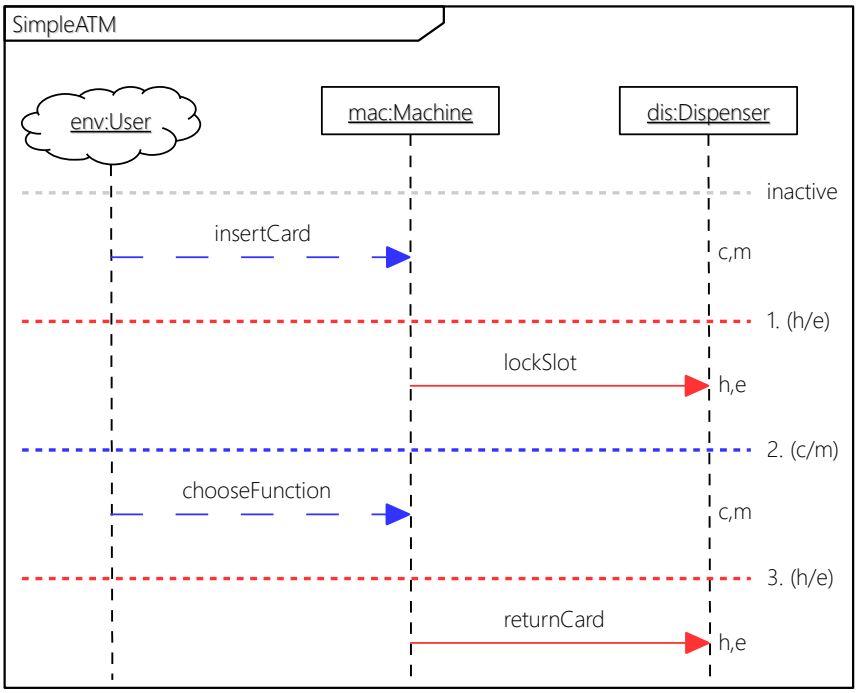


Abbildung 2.5: Beispielhafter Ablauf eines MSD.

In Abbildung 2.5 ist ein MSD Ablauf veranschaulicht. Zu Beginn befindet sich das MSD in einem Startzustand (*inactive*). Die graue gestrichelte Linie bedeutet hier, dass das MSD noch nicht initialisiert wurde. Bei

1. (h/e) wurde das MSD bereits initialisiert, durch die vorangegangene, erste Nachricht `insertCard`. Es wird nun als *active MSD* bezeichnet. Nachricht `lockSlot` wird als nächstes erwartet und ist nun *enabled*. Da die Nachricht hot executed ist, ist der cut ebenfalls hot executed. Im folgenden schreitet das MSD wieder einen Schritt voran und steht beim cut *2. (c/m)*. Die nächste aktivierte Nachricht ist `chooseFunction`. Da diese nun unifizierbar ist, schreitet der cut voran bis *3. (h/e)*. Hier muss nun Nachricht `returnCard` erfolgen und danach kann die aktive Instanz des MSDs `SimpleATM` geschlossen werden. Sollte an einem cut eine Nachricht auftreten, die nicht erwartet wird, weil z.B. eine andere Nachricht unifizierbar ist, tritt eine Verletzung des MSDs - auch *MSD Violation* genannt - auf. Im nun folgenden Abschnitt 2.3.4 werden diese MSD Violations behandelt.

2.3.4 MSD Violations

Während des Ablaufs eines MSD können drei verschiedene Arten von Verletzungen auftreten.

- *Safety Violation* oder *Hot Violation*: Diese tritt auf, wenn keine Nachricht mit execution kind hot *enabled* ist, stattdessen aber ein anderes Event auftritt, das unifizierbar mit einer Nachricht im gleichen MSD ist, die zur Zeit aber nicht enabled ist. In diesem Fall kann davon ausgegangen werden, dass ein schwerer Fehler aufgetreten ist. Hier würde eine Missachtung der Spezifikation vorliegen.

- *Cold Violation*: Analog zur *Safety Violation* tritt diese Verletzung an einem cold cut auf. Das aktive MSD wird beendet, nach Greenyer [7].

Die *Safety Violation* und *Cold Violation* treten also auf, wenn ein Event an einem cut auftritt, das nicht erwartet wird. Dies ist z.B. der Fall, wenn im MSD `SimpleATM` aus Abbildung 2.5 am cut *3. (h/e)* die Nachricht `chooseFunction` auftritt, anstatt von `returnCard`. Dies würde eine *Safety Violation* auslösen.

Des weiteren existiert noch eine dritte Art einer Verletzung.

- *Liveness Violation*: Tritt auf, wenn ein executed cut ausgeführt wird, das unifizierte Event aber niemals auftritt. Das System kann also nicht progressieren. Diese Art der Verletzung ist ebenso wie die *Safety Violation* nicht erlaubt. Im Gegensatz dazu, falls sich das System allerdings in einem cold cut befindet, kann es in diesem Zustand für immer verharren [7].

Der nächste Abschnitt führt die Assumption MSDs ein und definiert den Begriff der MSD Spezifikation.

2.3.5 Assumption MSDs und MSD Spezifikationen

Da Umweltereignisse unkontrollierbar sind, werden hier häufig Annahmen über das Verhalten der Umwelt gemacht (Zave and Jackson 1997 [19]). Im Fall des Geldautomaten könnte man beispielsweise die Annahme treffen, dass der Akteur nach Benutzung des Geldautomaten seine EC-Karte aus dem Gerät herausnimmt.

Die Art von Annahmen, die die Umwelt betreffen, können mit *Assumption MSDs* ausgedrückt werden. *Assumption MSDs* drücken also sowohl mögliches Verhalten der Umwelt aus, als auch wie die Umwelt auf das System reagiert. Die *Requirement MSDs* beschreiben das Verhalten des Systems wie es auf Umweltereignisse reagieren soll. Wird ein *Requirement MSD* verletzt, bedeutet das, dass ein Fehler im System aufgetreten ist, also dass die Spezifikation fehlerhaft oder dass ein Probleme bei der Modellierung aufgetreten ist. Eine Verletzung in einem *Assumption MSD* ist nicht gleichbedeutend damit, das ein Fehler im System vorliegen muss. *Assumption MSDs* enthalten zusätzlich die Annotation `<<Environment Assumption>>`. Eine Menge von Requirement MSDs und Assumption MSDs bilden zusammen eine *MSD Spezifikation*.

Für die Ausführung von *MSD Spezifikationen* ist die Annahme erforderlich, das das System ggf. schneller reagieren kann als die Umwelt. Das bedeutet, dass System ist in der Lage so viele Nachrichten wie nötig sind, innerhalb einer bestimmten Sequenz von Systemnachrichten zu versenden, bevor das nächste Umweltereignis eintritt. Konkret bedeutet dies, das System wartet auf eine Aktion der Umwelt. Tritt eine solche Nachricht auf, werden alle erlaubten und executed Nachrichten des Systems sequenziell abgearbeitet, die in der momentanen Anforderung erlaubt sind. Anschließend, wenn nur noch monitored Nachrichten vorhanden sind, geht das System wieder in einen Wartemodus über. Diese Annahme wird *synchrony hypothesis* genannt [4]. Somit ist eine Ausführung einer *MSD Spezifikation* erfüllbar genau dann, wenn:

1. sie von allen *Requirement MSDs* akzeptiert wird oder
2. sie wird von mindestens einem *Assumption MSD* nicht akzeptiert und
3. es ist keine unendliche Sequenz von Systemereignissen vorhanden.

Formal ausgedrückt bedeutet das. Sei R die Menge aller *Requirement MSDs*, also $R = \{r_0, \ldots, r_{n-1}\}$. Sei A die Menge aller *Assumption MSDs*, $A = \{a_0, \ldots, a_{n-1}\}$. π sei die Ausführung bzw. der Ablauf und die MSD Spezifikation ist definiert als $MS = (A, R)$. Dann gilt:

$$\pi \models MS$$
$$\Leftrightarrow \pi \models R, \forall r_i \in R \vee \exists a_i : a_i \in A \wedge \pi \not\models a_i.$$

2.3.6 Parameter und verbotene Nachrichten

Wie in 2.3.2 erwähnt, können MSD Nachrichten auch parametrisiert werden. D.h. das den MSD Nachrichten zusätzliche Parameter bestimmten Typs übergeben werden können. Des weiteren besteht die Möglichkeit, MSD Nachrichten zu definieren, die nicht auftreten dürfen. Das wird erreicht, indem man den Nachrichten die Annotation `<<forbidden>>` hinzufügt. Die Parametrisierung und verbotenen Nachrichten sollen in dieser Arbeit nicht weiter vertieft werden und sind nur der Vollständigkeit halber aufgeführt. Weiterführende Informationen zur Parametrisierung und verbotenen Nachrichten sind unter [10] und [7] zu finden.

2.4 Feature Diagrams

Im Folgenden werden Feature Diagrams eingeführt. Hierzu wird in Abschnitt 2.4.1 zuerst der Begriff *Feature* definiert und erklärt. Anschließend folgt der grafische Aufbau in 2.4.2. Darauf folgt ein Beispiel für ein Feature Diagram und dessen Erläuterung in 2.4.3. Zum Schluss wird in 2.4.4 noch eine formale Definition der Feature Diagrams vorgestellt.

2.4.1 Definition des Begriffs Feature

Der Begriff *Feature* wird in der Literatur unterschiedlich definiert. Kang et al. [14] definieren Feature als einen „hervorgehobenen oder charakteristischen für den Benutzer sichtbaren Aspekt oder sichtbares Qualitätsmerkmal eines Software-Systems oder Systems.“ Greenyer et al. [8] definieren Feature als „eine Einheit von Unterschieden zwischen Produkten, die den Interessengruppen [...] als natürlich erscheinen.“ In dieser Arbeit wird *Feature* definiert als Funktionalität oder Komponente, die in einer entsprechenden Variante (Produkt) einer Produktlinie vorhanden ist oder nicht.

Ein Produkt ist eine Zusammenstellung von Features, die die Mitgliedschaft in einer Produktlinie beschreiben. Bei einer Menge von Feature S kann ein Produkt als die Potenzmenge $\mathcal{P}(S)$ definiert werden, die alle Kombinationen an Features enthält. Jedoch ist nicht automatisch jede dieser Kombinationen ein gültiges Produkt. Einige dieser Features können voneinander abhängen. Andere wiederum schließen sich aus. Um eine gültige Menge an Feature-Kombinationen zu definieren, werden Feature Diagrams genutzt.

2.4.2 Grafischer Aufbau (Syntax)

Feature Diagrams (FDs) gehören zur Familie der Modellierungssprachen, die von Kang et al. [14] eingeführt werden. Hierbei werden FDs grafisch als eine

baumähnliche Struktur dargestellt, wobei die Wurzel immer das Konzept oder Hauptmerkmal genannt wird und das gesamte System widerspiegelt.

Die einzelnen Features werden als Knoten gezeichnet. Hierbei unterscheidet Schobbens et al. [18] primitive Features. Also Features, die für sich alleine stehen können und zusammengesetzte Features oder auch zerlegbare Features genannt. Primitive Features, so Schobbens et al., haben direkten Einfluss auf die Beschaffenheit des abgeleiteten Produkts. Die zusammengesetzten Features, die häufig nur für Zerlegungszwecke genutzt werden, drücken oft kein Feature an sich aus, so Gressi [10]. Somit beeinflusst die Wahl eines zusammengesetzten Features das resultierende Produkt nur unter Berücksichtigung der primitiven Features, die durch ihre deszendenten Knoten repräsentiert werden. In einem FD werden Knoten mit Kanten verbunden, um deren Beziehungen untereinander festzulegen. Sie können entweder zerlegbar oder bedingt (constraint) sein. Für zerlegbare Kanten existieren drei Typen:

AND AND-Zerlegung bedeutet, wenn ein Eltern-Feature gewählt wird, müssen alle Kind-Features ebenfalls gewählt werden.

OR OR-Zerlegung bedeutet, jede beliebige Kombination an Kind-Features kann gewählt werden. Allerdings mindestens eins.

XOR XOR-Zerlegung bedeutet, eine exklusive Alternativauswahl der Kind-Features, so dass genau eins gewählt werden kann.

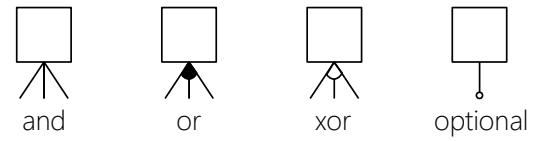
Zudem gibt es die Möglichkeit optionale Beziehungen festzulegen.

Optional Optional bedeutet, das ein bestimmtes Kind-Feature gewählt werden kann oder eben nicht.

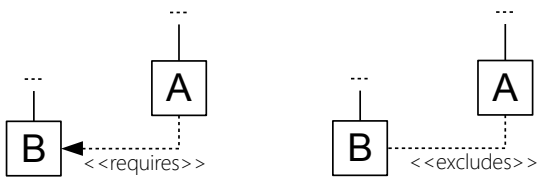
Unabhängig vom Zerlegungstyp kann ein Kind-Feature nur gewählt werden, wenn auch das Elternmerkmal gewählt ist. Des weiteren können zusätzliche Beziehungen ausgedrückt werden. Zum Beispiel kann ein Feature von einem anderen abhängen. Dieses wird mit der Annotation `<<requires>>` verdeutlicht. Oder zwei Features schließen sich gegenseitig aus wie z.B. die Wahl der Sitzbezüge in einem Auto. Entweder Leder oder Stoffbezug. Annotiert wird diese Kante dann mit `<<excludes>>`. Abbildung 2.6 zeigt die grafische Notation der verschiedenen Kantentypen.

2.4.3 Beispiel eines Feature Diagrams

Der Geldautomat kann ebenfalls mit Hilfe eines FDs visualisiert werden. Abbildung 2.7 zeigt das FD des Geldautomaten.



(a) Dekompositionskanten



(b) bedingte Kanten

Abbildung 2.6: Kantentypen in einem FD. In Anlehnung an [10]

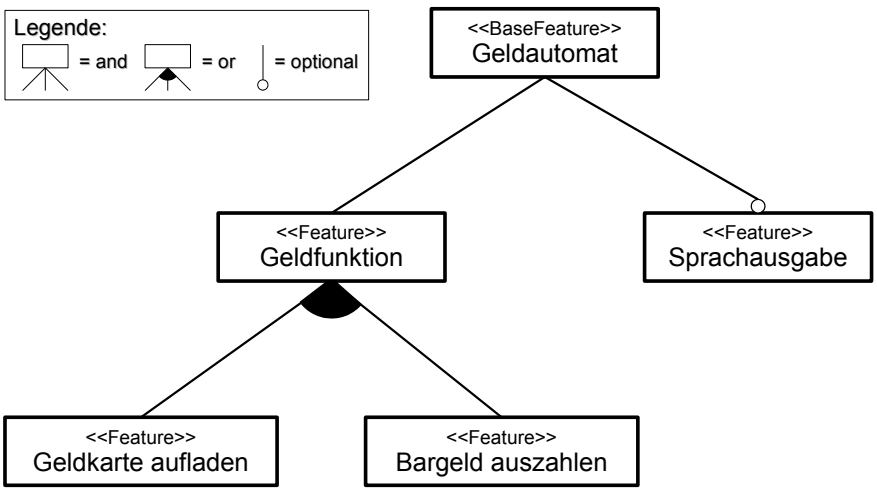


Abbildung 2.7: Feature Diagram des Geldautomaten

Die Wurzel bezeichnet das System `Geldautomat`. Dann kann aus zwei Features gewählt werden. Der `Geldfunktion` und der optionalen `Sprachausgabe`. Die Sprachausgabe kann also gewählt werden, muss aber nicht. Die `Geldfunktion` ist ein zerlegbares Feature und unterteilt sich in zwei weitere. Nämlich `Geldkarte aufladen` und `Bargeld auszahlen`. Der Dekompositionstyp der Kante ist *XOR*, d.h. es kann nur eines von beiden gewählt werden.

2.4.4 Formale Definition und Semantik

In diesem Abschnitt wird eine formale Definition der FDs in Anlehnung an Gressi [10] eingeführt.

Definition 2.4.1 (Feature Diagram)**.**
Ein FD ist ein Tupel $d = (N, P, r, \lambda, DE, CE)$, wobei

- N die Menge der Features ist;
- $P \subseteq N$ ist die Menge der primitiven Features;
- $r \in N$ ist das Wurzel-Feature. Nur r hat keine Eltern: $\forall n \in N.(\nexists\ n' \in N.n' \rightarrow n) \Leftrightarrow n = r$;
- $\lambda : N \rightarrow NT$ benennt jedes Features mit einem Operator aus NT, mit $NT = \{and,\ or,\ xor,\ opt\}$[1];
- $DE \subseteq N \times N$ ist die Menge der zerlegbaren Kanten, die einen Baum mit Wurzel r bilden müssen.
- $CE \subseteq N \times GCT \times N$ ist die Menge der bedingten Kanten, wobei $GCT = \{requires,\ excludes\}$.[2];

Definition 2.4.2 (Gültiges Produkt)**.**
Ein gültiges Produkt ist jedes $p \subseteq N$, das

- die Wurzel enthält: $r \in p$;
- erfüllbare Knotenbedeutungen aufweist: $\forall n \in p$, mit Kindern $s_1 \ldots s_k$ and $\lambda(n) = op_k$ dann muss op_k $(s_1 \in p,\ \ldots,\ s_k \in p)$ zu wahr evaluieren;
- alle grafischen Bedienungen erfüllt: $\forall(n_1,\ op_2,\ n_2) \in CE, op_2$ $(n_1 \in p,\ n_2 \in p)$ evaluiert zu wahr;
- falls s enthalten und s ist nicht die Wurzel, muss einer der Elternknoten n ebenfalls enthalten sein: $\forall s \in p.s \neq r\ \exists\ n \in p : n \rightarrow s$.

[1] NT (Node Type) ist die Menge der Zerlegungsoperatoren

[2] GCT (Graphical Constraint Type) ist die Menge der Typen an grafisch bedingten Kanten, die ein FD anbietet.

Die Semantik eines Feature Diagrams d, geschrieben als $[\![d]\!]_{FD}$, ist eine boolesche Formel, die die Menge an gültigen Produkten beschreibt. Z.B. die Kombination an Features, die die Bedingungen, ausgedrückt durch das FD, erfüllt. Solch eine Kombination ist eine Produktlinie.

Produktlinienspezifikation

Im Anschluss soll hier nun noch der Begriff der *Produktlinienspezifikation* definiert werden nach Cordy et al. [6]. Eine *Produktlinienspezifikation* besteht aus einer Menge an MSDs, der MSD Spezifikation und dem entsprechenden FD der Produktlinie.

2.5 Play-Out

Play-Out wird ein Algorithmus genannt, der eine operationale Interpretation von LSCs und MSDs ermöglicht. Ursprünglich wurde der Algorithmus von Harel and Marelly für LSCs definiert und später für MSDs erweitert [11], [12]. Dahinter steckt das Prinzip, anstatt einen *controller* für die Systemobjekte zu implementieren, werden die MSDs/LSCs nur interpretiert, so dass man daraus erfährt, welche Nachrichten die Systemobjekte senden können, müssen oder nicht senden dürfen nach einer bestimmten Sequenz von Events, so Greenyer [7].

Der Algorithmus für universale MSDs sieht wie folgt aus: Anfangs erhält er eine Menge von MSDs. Anschließend wartet der Algorithmus auf ein Umweltereignis. Ist dies der Fall, werden aktive Kopien der entsprechenden MSDs erzeugt, dessen erste Nachricht mit dem entsprechenden Event unifizierbar ist. An dieser Stelle wählt der Play-Out-Algorithmus nichtdeterministisch oder durch Benutzereingaben eine enabled Nachricht eines active MSDs und sendet diese. Sofern dieses nicht zu einer Safety-Violation führt. Durch diesen Vorgang wird solange iteriert bis alle aktiven MSDs beendet wurden oder keine Nachricht mehr übrig bleibt, die ohne das Auftreten einer Safety-Violation gesendet werden könnte. Ist letzteres der Fall terminiert der Algorithmus mit einem Fehler.

2.6 Featured Game Graph

Dieser Abschnitt behandelt Featured-Game-Graphen. Hierzu erfolgt zuerst eine kurze Einführung in Labelled Transition Systems 2.6.1 und Featured Transition Systems 2.6.2. Anschließend werden Game-Graphen in 2.6.3 besprochen und schließlich die Featured-Game-Graphen in 2.6.4. Die Definitionen hierfür stammen größtenteils aus [10].

2.6.1 Labelled Transition Systems

Labelled Transition Systems (LTS) sind ein häufig genutztes Modell, um das Verhalten eines einzelnen Systems darzustellen (Baier und Katoen [2]). Ein LTS ist ein gerichteter Graph, indem die Knoten Zustände (*states*) und die Transitionen die Kanten zwischen den Zuständen repräsentieren. Transitionen markieren hierbei einen Zustandsübergang. Eine Menge von Startzuständen stellt die mögliche Startkonfiguration zum Start dar. Transitionen werden hierbei i.d.R. nach dem Zustandsübergang benannt. Zustände werden mit atomaren Bezeichnern versehen. Formal werden LTS wie folgt definiert:

Definition 2.6.1 (Labelled Transition System)**.**
Ein LTS ist ein tuple $lts = (S,\ Act,\ trans,\ I,\ AP,\ L)$, wobei

- S ist die Menge an Zuständen
- Act ist die Menge der Aktionen
- $trans \subseteq S \times Act \times S$ ist die Menge der Transitionen mit $(s_1,\ \alpha,\ s_2) \in trans$
- $I \subseteq S$ ist die Menge der Startzustände
- AP die Menge der atomaren Bezeichner
- $L : S \to 2^{AP}$ ist die Labelling-Funktion

2.6.2 Featured Transition Systems

Featured Transition Systems (FTS) erweitern das Konzept der LTS für Produktlinien, wobei *Features* zum vorrangigen Konzept gemacht werden. Bevor eine formale Definition der FTS folgt, soll das Grundkonzept anhand eines Beispiels in Anlehnung an [10] veranschaulicht werden.

Abbildung 2.8(a) und Abbildung 2.8(b) zeigen zwei LTS, die das Verhalten eines Ampelsystems widerspiegeln. Variante (a) zeigt eine Ampel, die zwischen rot und grün umschalten kann. Version (b) stellt eine Erweiterung von (a) da. Hier wird zusätzlich gelb vor Umschalten auf rot angezeigt. Die Produktlinie dieser Varianten ist nun im FD in Abbildung 2.8(c) dargestellt. Die zusätzliche Ampelfarbe gelb wird hier durch das optionale Feature `Y` ausgedrückt. Damit nun das Feature `Y` berücksichtigt werden kann, ist es erforderlich, zusätzliche Zustände und Transitionen hinzuzufügen. Hierzu können FTS genutzt werden, die das Systemverhalten und Features in Beziehung setzen. Erreicht wird dieses durch boolesche Bedingungen an den Transitionen mit einem „/“ als Trennzeichen.

Abbildung 2.8(d) zeigt das FTS für das Ampelsystem als ein Modell. Die Transition ①→② ist sowohl mit einem Bezeichner `yellow` als auch

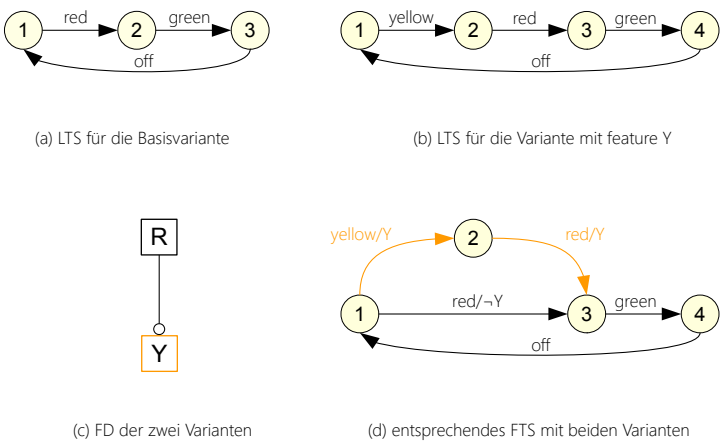


Abbildung 2.8: FTS des traffic light controllers in Anlehung an [10]

mit dem Feature `Y` versehen. `Y` bedeutet hier, dass nur Produkte mit Feature `Y` diese Transition nehmen können. Zusätzliche Features fügen nicht nur Zustände und Transitionen hinzu, sie können sie genauso entfernen. So entfernt das Feature `Y` die (1)→(3) Transition. Dies wird erreicht durch die Bezeichnung ¬`Y`. Formal werden FTS wie folgt definiert:

Definition 2.6.2 (Featured Transition System)**.**
Ein FTS ist ein tuple $fts = (S,\ Act,\ trans,\ I,\ AP,\ L,\ d,\ \gamma)$, wobei

- $S,\ Act,\ trans,\ I,\ AP,\ L$ sind in Definition 2.6.1 definiert;
- d ist in Definition 3.1.1 definiert;
- $\gamma : trans \to (F) \to \{\top,\ \bot\}$ ist eine totale Funktion, die jede Transition mit entsprechender Feature expression bezeichnet.

2.6.3 Game Graph

Der in dieser Arbeit verwendete Ansatz zur Synthese für szenariobasierte Spezifikationen von Produktlinien kann angesehen werden als Problem des Findens einer Strategie in einem unendlichen Spiel. Eine Partei des Spiels ist das System, das gegen die Umwelt als andere Partei spielt. Das System muss hierbei immer als Gewinner hervorgehen.

Die meisten Spiele terminieren sobald ein Gewinner feststeht. Zum Beispiel beim Schach (König wurde geschlagen) oder Mühle (weniger als drei Steine übrig). Reaktive Systeme terminieren allerdings nicht unbedingt. Daher müssen sie „immer wieder gewinnen“, bzw. korrekter gesagt: Das System gewinnt, wenn es garantieren kann, immer wieder einen Zielzustand zu erreichen.

Der Algorithmus dieser Arbeit basiert auf solchen *Büchi-Spielen*. Benannt nach dem Schweizer Logiker und Mathematiker Julius Richard Büchi. *Büchi-Spiele* sind Spiele, bei denen unendlich oft ein Zustand mit gegebenen Eigenschaften erreicht werden muss. Die Struktur dieses Spiels ist ein Graph. Eine sogenannter *Game-Graph*. Die Knoten werden hier *Zustände* oder auch *states* genannt. Der *Game-Graph* basiert auf LTS (2.6.1), wobei jeder *state* genau einem *cut* in jedem MSD zugeordnet werden kann, das Teil der Spezifikation ist. Der Startzustand entspricht keinem aktiven MSD. Ein *Spielzug* des Systems respektive der Umwelt stimmt mit einer Nachricht überein, die vom System respektive der Umwelt gesendet wird. Die Kanten oder auch *Transitionen* repräsentieren hierbei einen erlaubten Spielzug für einen gegebenen Zustand. Bezeichnet werden die Transitionen mit dem Event, das sie repräsentieren. Die Transitionen werden in kontrollierbar (Linien mit Pfeilende) und unkontrollierbar (gestrichelte Linien mit Pfeilende) unterteilt. Kontrollierbare Transitionen stellen Systemereignisse dar. Unkontrollierbare Transitionen die Umweltereignisse, da das System keine Möglichkeit hat, Umweltereignisse zu erzwingen. Eine Strategie kann als *gewinnend* (*winning*) angesehen werden, wenn garantiert werden kann, dass das System immer gewinnt, unabhängig davon was die Umwelt tut.

Die Zielzustände in einem Game-Graphen werden *goal states* genannt. Um nun zu sagen, dass eine Produktlinienspezifikation widerspruchsfrei ist, also dass ein bestimmtes Produkt realisierbar ist, muss sichergestellt werden, dass ein *goal state* unendlich oft erreicht werden kann, unabhängig davon was die Umwelt tut. Zu erwähnen ist, dass nicht zwangsläufig jeder *goal state* auch *winning* ist. Ein *goal state* besitzt die folgenden Bedienungen:

1. Es trat keine Safety Violation in einem Requirements MSD auf und
2. es sind keine active events mehr in einem active Requirements MSD oder
3. eine Safety Violation trat in einem Assumption MSD auf oder
4. es ist noch mind. ein active event in einem active Assumption MSD vorhanden.

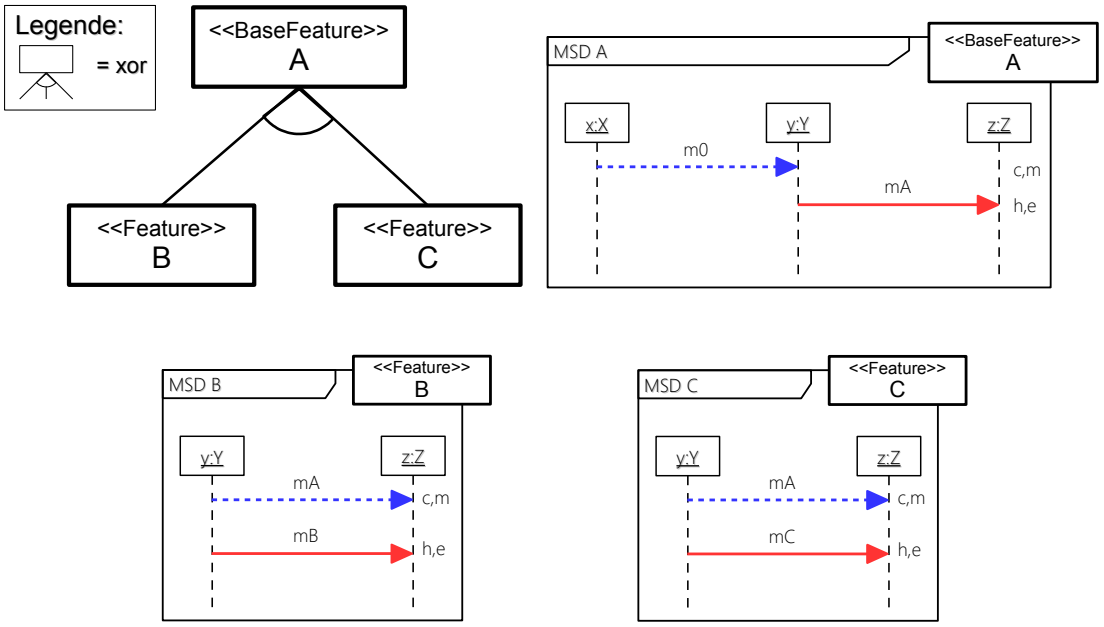
2.6.4 Featured Game Graph

Ein *Featured Game-Graph* (FGG) ist eine Erweiterung eines Game-Graphen. Die Transitionen werden hier als *featured transitions* bezeichnet und tragen neben dem Ereignisnamen zusätzlich eine *feature expression*. Aufgrund der Tatsache, dass unterschiedliche Features an einem Spielzug beteiligt sein können, die wiederum unterschiedliche MSDs aktivieren können, kann ein state in einem FGG mit unterschiedlichen *featured transitions* zu verschiedenen Nachfolgeknoten führen. Sollte beim Voranschreiten eines

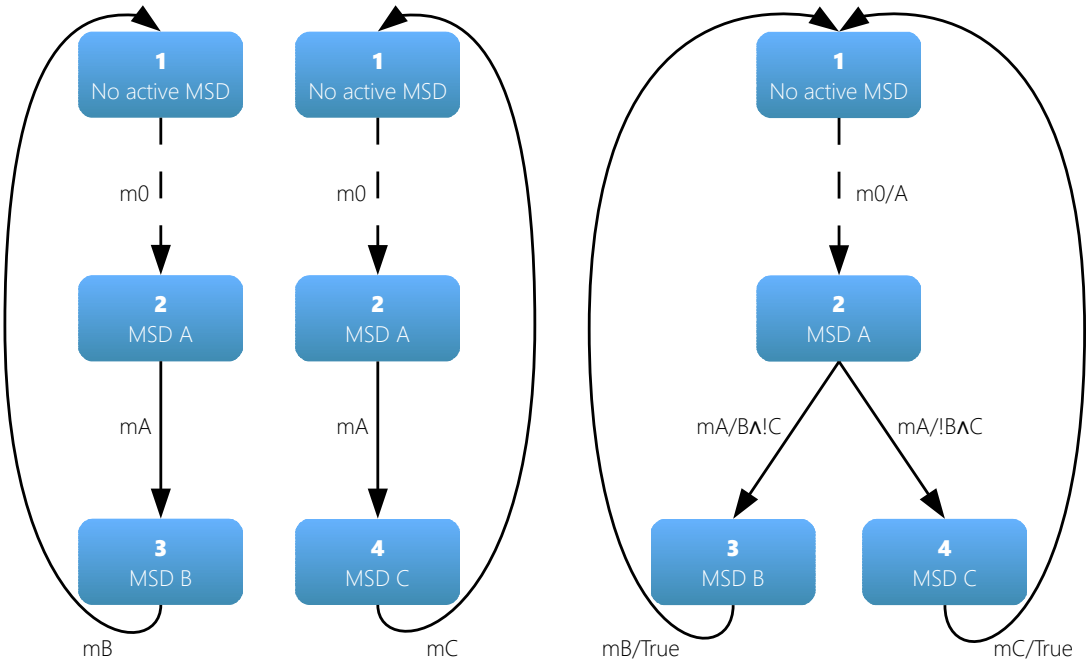
MSDs nach einem cut kein Neues aktiviert werden, dass nicht schon im aktuellen Zustand aktiv war, wird die *feature expression* der Transition mit einem `<<True>>` bezeichnet.

In Anlehnung an [10] ist ein Beispiel hierfür in Abbildung 2.9 zu finden. Abbildung 2.9(a) zeigt die Produktlinienspezifikation für die Produkte `{{A,B}{A,C}`. Die daraus resultierenden zwei Game-Graphen sind in Abbildung 2.9(b) zu sehen. In diesem Beispiel korrespondiert das Event `<<mA>>` zu zwei verschiedenen Transitionen, je nach Produktzugehörigkeit. Also je nachdem, ob Feature `<<B>>` oder `<<C>>` enthalten sind.

Beim Featured-Game-Graphen in Abbildung 2.9(c) ist diese Information mit in der Transition `<<mA>>` kodiert. Die feature expression lauten hier `<<B∧!C>>` und `<<!B∧C>>`. Die unkontrollierbare Transition `<<m0>>` von Zustand (1) zu Zustand (2) trägt die feature expression `<<A>>`, weil das MSD in Feature `A` vom Event `<<m0>>` aktiviert wird. Die übrigen featured Transitionen sind hier mit `True` bezeichnet.



(a) PL-spezifikation für zwei Produkte {{A,B} {A,C}}



(b) Zwei Game-Graphen

(c) Ein Featured-Game-

Abbildung 2.9: Beispiel eines FGG in Anlehung an [10]

2.7 ScenarioTools

ScenarioTools[3] ist eine Eclipse-basierte Werkzeugumgebung. Sie unterstützt einen bei der Modellierung und Analyse szenariobasierter Spezifikationen. Die Modellierung erfolgt hierbei über einen grafischen UML-Editor. Somit können MSD Spezifikationen und FDs erstellt werden. Diese können anschließend über verschiedene Play-Out-Varianten simuliert werden. Die Realisierbarkeit einer Spezifikation kann durch Synthese überprüft werden. Hierbei wird ein *Controller* (Steuerung) des Systems erzeugt. Gibt es einen solchen Controller oder präziser ausgedrückt, konnte ein solcher Controller synthetisiert werden, ist die Spezifikation realisierbar. Anschließend können die Controller in das PDF-Format exportiert werden, zur weiteren Unterstützung oder Orientierungshilfe.

2.8 Stand der Technik

Dieser Abschnitt soll einen Überblick über den aktuellen Stand der Technik und verwandte Arbeiten geben. Mit „aktueller Stand der Technik“ ist hier folgendes gemeint. Der in dieser Arbeit vorgestellte Algorithmus basiert auf der Arbeit „Featured Synthesis of Scenario-Based Software Product Line Specifications“ von Gressi [10]. Dazu werden in Abschnitt 2.8.2 Details diskutiert, die zum weiteren Verständnis für Kapitel 3 notwendig sind. Abschnitt 2.8.1 befasst sich, wie zu Anfang dieses Abschnitts erwähnt, mit verwandten Arbeiten.

2.8.1 Verwandte Arbeiten

Löding et al. stellen in „Synthesis of Open Reactive Systems from Scenario-Based Specifications“ einen Ansatz vor, mit Hilfe von LSCs ein szenariobasierte Spezifikationen zu überprüfen. Hierbei wird ein spieltheoretischer Ansatz verfolgt. Es wird versucht mittels Synthese eine Strategie zu finden, in der das System sicherstellen kann, dass die Spezifikation eingehalten wird. Sollte die Spezifikation allerdings nicht implementierbar sein, soll eine Gegenstrategie für die Umwelt abgeleitet werden, die das System verlieren lässt [5].

Harel und Segall zeigen in „Synthesis from scenario-based specifications“ ebenfalls einen Ansatz, der ebenfalls mit LSCs arbeitet. Hier wird ein Spiel zwischen System und Umwelt gespielt. Gibt es für das System eine gewinnende Strategie, so wurde die Spezifikation korrekt umgesetzt [13].

Greenyer et al. haben in „Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications“ einen Ansatz vorgestellt, Produktlinien mit Hilfe von MSDs und FDs zu beschreiben und anschließend *on-*

[3] `http://scenariotools.org`

the-fly einen Controller zu synthetisieren. Dieser Algorithmus ist ebenfalls spieltheoretischer Natur und basiert auf Büchi-Spielen. *On-the-fly* bedeutet hier, dass die zugrundeliegende Spielstruktur - ein *Game-Graph* (2.6.3) - nicht komplett untersucht wird, sondern lediglich nur gewisse Teile, die zu einer gewinnenden Strategie führen. Das wird algorithmisch mit einer *required goals map* gelöst, die für bestimmte Zustände, die Menge zu erreichender Zielzustände zwischenspeichert, für die garantiert werden kann, dass diese *winning* sind [8].

2.8.2 Featured Synthese

Im Folgenden soll nun die „Featured Synthesis of Scenario-Based Software Product Line Specifications“ von Gressi [10] besprochen werden. Zum einfacheren Verständnis gliedert sich dieser Abschnitt in drei Fragestellungen.

Was macht die Featured Synthese?

Der Featured Synthese liegt eine Produktlinienspezifikation zugrunde, anhand dessen entschieden werden soll, für welche daraus resultierenden Produkte es eine Strategie gibt, für die garantiert werden kann, dass sie das Büchi-Spiel gewinnen. Diese *winning products*, bestehend aus einer validen Kombination der unterschiedlichen Features des FDs, sind dann also realisierbare, die Spezifikation nicht verletzende Produkte der Produktlinie.

Wie arbeitet die Featured Synthese?

Zuallererst wird aus dem FD der Produktlinienspezifikation die Menge der gültigen Produkte abgeleitet. Produkte, dessen Features sich gegenseitig ausschließen oder sonstige Bedingungen der FD-Syntax verletzen, werden gar nicht erst beachtet. Anschließend wird intern der FGG aufgebaut und der Algorithmus exploriert den Graphen. Angefangen beim Startzustand. Hierfür werden jeweils die ausgehenden Transitionen eines Zustandes genutzt, um den nächsten Zustand zu untersuchen. Falls einer der untersuchten Zustände ein *goal state* ist, wird dieser gespeichert. An dieser Stelle findet eine Rückwärtspropagierung statt. D.h. es wird jetzt geprüft, ob für alle seine Vorgänger ebenfalls garantiert werden kann, dass diese ebenfalls gewinnen können.

Was soll besser werden?

Der Algorithmus soll hinsichtlich seiner Geschwindigkeit optimiert werden. Dazu soll er ähnlich wie in [8] *on-the-fly* arbeiten. D.h. es soll nach Möglichkeit nicht der gesamte FGG exploriert werden und die bereits explorierten Zustände, für die garantiert werden kann, dass sie zu *winning goal states* führen, zwischengespeichert werden. Intuitiv liegt die Vermutung

nahe, dass die *featured on-the-fly Synthese* schneller ist, da nur ein Teil des FGGs im Idealfall exploriert werden muss. Ob sich das später wirklich in der Praxis bestätigen wird, soll in den folgenden Kapiteln untersucht werden.

Kapitel 3

On-the-fly Synthese

Nach einer kurzen Einleitung in 3.1 und einem Überblick über die Struktur des Algorithmus in Abschnitt 3.2, wird der Synthesealgorithmus anhand eines FG-Graphen exemplarisch in 3.3 erklärt. Am Ende dieses Kapitels soll ein Gegenbeispiel in Abschnitt 3.4 gezeigt werden, bei dem der Algorithmus keine realisierbaren Produkte findet.

3.1 Einleitung

Der Synthesealgorithmus basiert wie in Kapitel 2.8.2 erwähnt, auf der featured Synthese von Gressi [10], sowie ersten Überlegungen von Joel Greenyer und Maxime Cordy. Ziel des *on-the-fly* Ansatzes ist es, die Geschwindigkeit der Synthese zu erhöhen. Das soll dadurch erreicht werden, dass der Algorithmus sich merkt, für welche *states* er *winning goal states* erreichen kann und somit nicht unbedingt den gesamten FGG explorieren muss.

3.1.1 Produktableitung

Zu Beginn leitet der Algorithmus die validen Produkte aus dem Feature Diagram unter Berücksichtigung der FD-Semantik ab. Diese abgeleiteten Produkte müssen noch nicht zwangsläufig alle gewinnenden Produkte darstellen, denn Ihre Spezifikation kann noch Widersprüche enthalten. Sie bilden lediglich die Basis, auf der der Algorithmus anschließend gewinnende Produkte synthetisieren kann. Die Einzelheiten zur algorithmischen Ableitung sind unter [10] (S.43ff) zu finden und sollen hier nicht weiter vertieft werden. Für das Geldautomatenbeispiel würde die Ableitung aus dem FD in Abbildung 2.7 wie folgt aussehen:

$$(GA \wedge SA \wedge GF \wedge GKL) \vee (GA \wedge \neg SA \wedge GF \wedge GKL)$$
$$\vee (GA \wedge SA \wedge GF \wedge BA) \vee (GA \wedge \neg SA \wedge GF \wedge BA)$$

Wobei die Kürzel hier wie folgt definiert sind:

- $GA \mathrel{\widehat{=}}$ Geldautomat
- $SA \mathrel{\widehat{=}}$ Sprachausgabe
- $GF \mathrel{\widehat{=}}$ Geldfunktion
- $GKL \mathrel{\widehat{=}}$ Geldkarte aufladen
- $BA \mathrel{\widehat{=}}$ Bargeld auszahlen

3.1.2 Post-Algorithmus

Die Algorithmen 2, 4 und 5 bedienen sich einer Funktion $Post(s)$. Diese wird in [10] (S.45ff) ausführlich erläutert. An dieser Stelle soll nur ein Überblick gegeben werden, was die Ein- bzw. Ausgabe dieser Funktion ist. Die Funktion enthält als einzigen Eingabeparameter einen *state*, für den sie alle ausgehenden Transitionen zurückgibt.

Definition 3.1.1 (Ausgabe Post(s))**.**
Eine Transition ist in diesem Fall ein Tupel $e = (s,\ \sigma,\ s',\ \gamma)$, wobei

- s der Startzustand oder auch *source state* ist,
- σ der Bezeichner der Transition oder auch *label* ist,
- s' der Zielzustand oder auch *target state* ist,
- γ die *feature expression* der Transition ist.

3.2 Struktur des Algorithmus

Die FOTF-Synthese gliedert sich im Wesentlichen in zwei Teilalgorithmen. Zum einen in das Büchi-Spiel, das durch den Algorithmus *FOTFB* 1 gegeben ist und zum anderen in den Reachability-Algorithmus *FOTFR* 2, 3. Die Algorithmen sollen hier nicht Zeile für Zeile bearbeitet und erklärt werden. Dieser Teil soll lediglich einen Überblick über die Struktur der FOTF-Synthese geben. In Abschnitt 3.3 soll der Algorithmus dann anhand eines Beispiel eingehender untersucht und erläutert werden.

3.2.1 Featured Büchi

FOTFB prüft für einen gegebenen Startzustand s_0, ob das System garantieren kann, mindestens einen goal state unendlich oft zu besuchen. Hierzu wird zuerst in Zeile 7 die Reachability-Prozedur *FOTFR* aufgerufen, um den FGG auf goal states zu untersuchen. Anschließend wird für jeden gefundenen goal state g, für den noch nicht entschieden ist, ob dieser *winning* oder *losing* ist, die Reachability-Prozedur erneut aufgerufen (Zeile 10).

Algorithm 1 Featured On-the-fly Buechi

```
1:  global variables:
2:  Set<State> Goal
3:  Map<State, fexpr> loseBuechi
4:  Map<State, fexpr> WinR
5:  Map<State, Map<State, fexpr> > ReqG

6:  procedure FOTFB(s_0 : State, vp : fexpr)
7:      FOTFR(s_0, vp);
8:      while (∃g ∈ Goal | ¬(vp ⇒ WinR[g] ∨ loseBuechi[g])) do
9:          undecidedProducts ← vp ∧ ¬WinR[g] ∧ ¬loseBuechi[g];
10:         FOTFR(g, undecidedProducts);
11:     end while
12:     return WinR[s_0];
13: end procedure
```

3.2.2 Reachability

Der Reachability-Algorithmus 2 prüft solange für einen gegebenen Zustand s_0, bis entschieden ist, ob dieser *winning*, also für s_0 ein goal state erreicht werden kann oder dieser *losing* ist.

Dazu werden für s_0 zuerst alle ausgehenden Transitionen auf dem *Waiting*-Stack gespeichert (Zeile 3). Dann wird solange iteriert (Zeile 5), bis für s_0 entschieden ist, ob dieser losing oder winning ist. Dafür werden die ausgehenden Transitionen von s_0 untersucht und geschaut, ob die Zielzustände s' bereits besucht worden sind (Zeile 7). Dieser Vorgang wird als *Vorwärtsexploration* bezeichnet. Ist dies nicht der Fall, werden Sie markiert und es wird geprüft, ob s' möglicherweise ein goal state ist. Ist das der Fall, wird s' der *Goal*-map hinzugefügt, die sich alle besuchten goal states merkt. Anschließend wird diese Transition abermals auf den *Waiting*-Stack gelegt, zur *Rückwärtsreevaluierung*. Dies geschieht ebenfalls, falls s' noch ein unentschiedenes Produkt ist. Unentschiedene Produkte zeichnen sich dadurch aus, das für sie noch keine eindeutige Aussage gemacht werden kann, ob die entsprechenden Zustände winning oder losing sind.

Sollte s' kein goal state sein, werden seine Nachfolger exploriert. Wurde s' bereits als besucht markiert, wird sein Vorgänger, der Zustand s, rückwärts reevaluiert (Zeile 27ff., Algorithmus 3).

Hierbei wird unterschieden zwischen kontrollierbaren und unkontrollierbaren Transitionen (Zeile 27). Für unkontrollierbare Transitionen muss sichergestellt werden, dass jede dieser Transitionen zu einem winning oder goal state führt. Die beiden Prozeduren *ProcessControllableTransitions* 4 und *ProcessUncontrollableTransitions* 5 bauen schließlich die *ReqG*-map (required goals map) auf, die eine Menge von Zuständen auf die Menge der

Algorithm 2 Featured On-the-fly Reachability Part 1

```
1: procedure FOTFR(s_0, undecidedProducts)
2:     Visited ← {s_0};
3:     Waiting ← {e = (s, σ, s', γ) | e = Post(s_0)};
4:     Depend[s_0] ← ∅;
5:     while (¬(undecidedProducts ⇒ WinR[s_0] ∨ loseBuechi[s_0])) do
6:         Pop e = (s, σ, s', γ) from Waiting;
7:         if (s' ∈ Visited) then
8:             Visited ← Visited ∪ {s'};
9:             Depend[s'] ← {e};
10:            if (isGoal(s')) then
11:                Goal ← Goal ∪ {s'};
12:            else
13:                for each ((G', fexpr) ∈ ReqG[s'], G' ⊆ Goal) do
14:                    for each (g ∈ G') do
15:                        ReqG[s'][G'] ← ReqG[s'][G'] ∧ ¬loseBuechi[g];
16:                    end for
17:                end for
18:                WinR[s'] ← ∀(G', fexpr)ReqG[s'] : ∨ fexpr;
19:            end if
20:            if (isGoal(s') ∨ (undecidedProducts ⇒ WinR[s'] ∨
    loseBuechi[s'])) then
21:                Waiting ← Waiting ∪ e;
22:            else
23:                Waiting ← Waiting∪ {e = (s', σ, s'', γ) | e = Post(s')};
24:            end if
25:        else
26:                ▷ Backward propagation for readability reasons in part 2
```

erreichenbaren goal states mit samt den entsprechenden feature expressions abbildet. Zudem wird die *WinR*-map für die entsprechenden Zustände abgeleitet. Diese Map speichert für einen gegebenen Zustand s_i die entsprechenden gewinnenden feature expressions (*winning products*). Konträr zur *WinR*-map speichert die *loseBüchi*-map alle verlierenden Produkte für einen Zustand s_i.

Algorithm 3 Featured On-the-fly Reachability Part 2

```
27:          if (isControllable(s)) then
28:              ProcessControllableTransitions(s)
29:          else
30:              ProcessUncontrollableTransitions(s)
31:          end if
32:          if („s has no unvisited successors“) then
33:              loseBuechi[s] ← ¬WinR[s];
34:          else
35:
```

$$loseBuechi[s] \leftarrow \bigwedge_{(s,\ \sigma,\ s',\ \gamma) \in Post_c(s)}^{controllable} \gamma \wedge loseBuechi[s'] \vee \bigvee_{(s,\ \sigma,\ s',\ \gamma) \in Post_u(s)}^{uncontrollable} \gamma \wedge loseBuechi[s'];$$

```
36:          end if
37:          if (undecidedProducts ⇒ WinR[s] ∨ loseBuechi[s]) then
38:              Waiting ← Waiting ∪ Depend[s];
39:          end if
40:      end if
41:  end while
42:  return WinR[s_0];
43: end procedure
```

Algorithm 4 Process controllable transitions

```
1: procedure ProcessControllableTransitions(s)
2:     WinRProducts_s ← false;
3:     for each ((s, σ, s', γ) ∈ Post(s)) do
4:         var WinRProducts_s' : Fexpr;
5:         if (isGoal(s')) then
6:             WinRProducts_s' ← ¬loseBuechi[s'];
7:             ReqG[s] ← ReqG[s] ∪ ({s'}, WinRProducts_s' ∧ γ ∧ ¬WinRProducts_s);
8:         else
9:             for each ((G', fexpr) ∈ ReqG[s'], G' ⊆ Goal) do
10:                for each (g ∈ G') do
11:                    ReqG[s][G'] ← ReqG[s][G'] ∪ ReqG[s'][G'] ∧ γ ∧ ¬WinRProducts_s;
12:                end for
13:            end for
14:        end if
15:        WinRProducts_s ← WinRProducts_s ∨ (γ ∨ (WinRProducts_s'));
16:        WinR[s] ← WinRProducts_s;
17:        if (WinR[s] ⇒ undecidedProducts) then
18:            break;
19:        end if
20:    end for
21: end procedure
```

Algorithm 5 Process uncontrollable transitions

1: **procedure** ProcessUncontrollableTransitions(s)
2: $WinRProducts_s \leftarrow false$;
3: **for each** $((s,\ \sigma,\ s',\ \gamma) \in Post(s))$ **do**
4: var $Win_{s'}$: Fexpr;
5: **if** (isGoal(s')) **then**
6: $Win_{s'} \leftarrow \neg loseBuechi[s']$;
7: $ReqG[s] \leftarrow ReqG[s] \cup (\{s'\},\ Win_{s'} \wedge \gamma \wedge \neg WinRProducts_s)$;
8: $WinRProducts_s \leftarrow WinRProducts_s \vee (\gamma \wedge Win_{s'})$;
9: **else**
10: **if** $(ReqG[s] = \varnothing)$ **then**
11: **if** $(ReqG[s'] = \varnothing)$ **then**
12: **continue**;
13: **end if**
14: $Win_{s'} \leftarrow \forall (G',\ fexpr) \in ReqG[s'] :\ \vee\ fexpr$;
15: **for each** $((G',\ fexpr) \in ReqG[s'],\ G' \subseteq Goal)$ **do**
16: **for each** $(g \in G')$ **do**
17: $ReqG[s][G'] \leftarrow ReqG[s'][G'] \wedge \gamma \wedge Win_{s'}$;
18: **end for**
19: **end for**
20: $WinRProducts_s \leftarrow WinRProducts_s \vee (\gamma \wedge Win_{s'})$;
21: **else if** $(ReqG[s'] = \varnothing)$ **then**
22: $Win_{s'} \leftarrow \forall (G',\ fexpr) \in ReqG[s] :\ \vee\ fexpr$;
23: **for each** $((G',\ fexpr) \in ReqG[s],\ G' \subseteq Goal)$ **do**
24: **for each** $(g \in G')$ **do**
25: $ReqG[s][G'] \leftarrow ReqG[s][G'] \wedge \neg\gamma \wedge Win_{s'}$;
26: **end for**
27: **end for**
28: $WinRProducts_s \leftarrow WinRProducts_s \vee (\neg\gamma \wedge Win_{s'})$;
29: **else**
30: **assert** $ReqG[s'] \neq \varnothing$;
31: $Win_{s'} \leftarrow \forall (G',\ fexpr) \in ReqG[s'] :\ \vee\ fexpr$;
32: **for each** $((G',\ fexpr) \in ReqG[s'],\ G' \subseteq Goal)$ **do**
33: **for each** $(g \in G')$ **do**
34: $ReqG[s][G'] \leftarrow ReqG[s][G'] \cup ReqG[s'][G'] \wedge \gamma \wedge Win_{s'}$;
35: **end for**
36: **end for**
37: $WinRProducts_s \leftarrow WinRProducts_s \vee (\gamma \wedge Win_{s'})$;
38: **end if**
39: **end if**
40: $WinR[s] \leftarrow WinRProducts_s$;
41: **end for**
42: **end procedure**

3.3 Beispiel

Im Folgenden wird der Algorithmus nun anhand eines FGGs exemplarisch erklärt und mit entsprechenden Abbildungen veranschaulicht.

Als erstes wird $FOTFB(s_0,\ \gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3)$ aufgerufen. Nun erfolgt der erste Reachability-Durchlauf für den Startzustand (s_0) (Zeile 7, Algorithmus 1). Dort werden alle ausgehenden Transitionen von (s_0) (blau markiert) auf den *Waiting*-Stack gelegt wie in Abbildung 3.1 zu sehen.

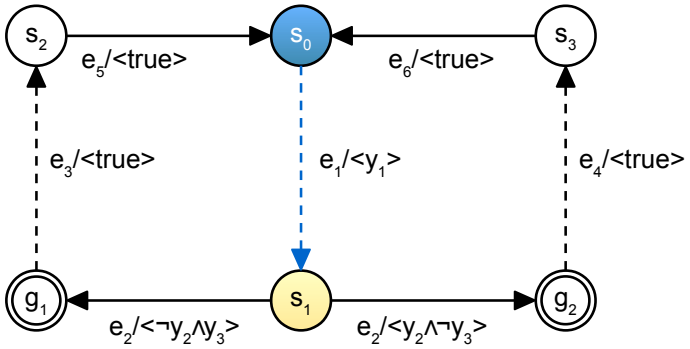


Abbildung 3.1: Synthese Algorithmus Beispiel 1 - Schritt 1

Der Zielzustand (s_1) (gelb markiert) wurde noch nicht besucht und nun markiert (Zeile 7, Algorithmus 2). Es handelt sich nicht um einen goal state. Der else-Zweig in Zeile 13-18 hat keinen Einfluss, da noch kein Eintrag in $ReqG\,[s_1]$ vorliegt. Stattdessen werden die nachfolgenden Transitionen e_2 auf den *Waiting*-Stack gelegt, wie in Abbildung 3.2 zu sehen.

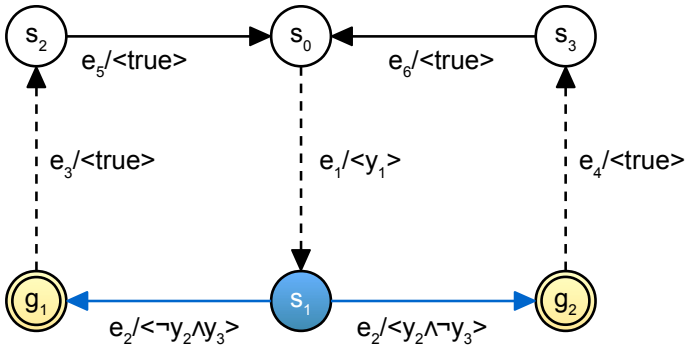


Abbildung 3.2: Synthese Algorithmus Beispiel 1 - Schritt 2

(g_2) ist ein goal state, der noch nicht besucht wurde. Daher wird in Zeile 21 diese Transition wieder auf den Stack gelegt für die Rückwärtsevaluierung. Anschließend wird diese Transition wieder vom Stack genommen und in Zeile 30 von Algorithmus 3 die Funktion $ProcessControllableTransitions(s_1)$

aufgerufen. Hier wird nun die *required goals map* für (s_1) und die *winning map* gefüllt. Die globalen Variablen sind nun wie folgt belegt:

```
1  Goals = {g1, g2}
2  WinR = {(s1, ¬γ2 ∧ γ3 ∨ ∧γ2 ∧ ¬γ3)}
3  loseBuechi = {(s1, false)}
4  ReqG[s1] = {
5      ({g1}, γ2 ∧ ¬γ3),
6      ({g2}, ¬γ2 ∧ γ3)}
```

Jetzt erfolgt eine Rückpropagierung (Zeile 38, Algorithmus 3) wie in Abbildung 3.3 gezeigt wird. Hierbei wird das Transitionstupel $(s_0, e_1, s_1, \gamma_1)$ aus der *Depend*-Map für (s_1) wieder auf den Stack gelegt. Also die eingehende Transition e_1 für den Zustand (s_1).

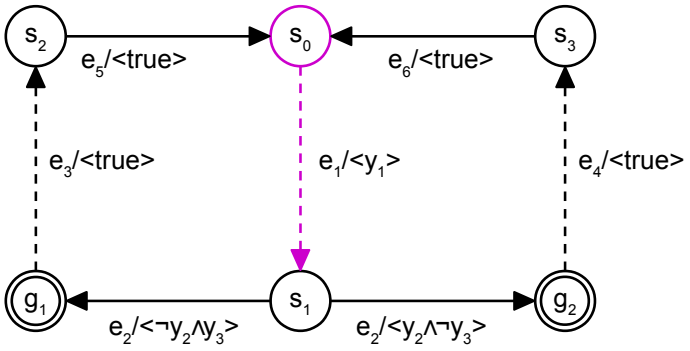


Abbildung 3.3: Synthese Algorithmus Beispiel 1 - Rückpropagierung

Der Zustand (s_1) ist bereits als besucht markiert und es wird *ProcessUncontrollableTransitions*(s_0) aufgerufen, da e_1 eine unkontrollierbare Transition ist. Die globalen Variablen sind danach wie folgt belegt:

```
1  Goals = {g1, g2}
2  WinR = {(s0, γ1 ∧ ¬γ2 ∧ γ3 ∨ γ1 ∧ γ2 ∧ ¬γ3),
3          (s1, ¬γ2 ∧ γ3 ∨ ∧γ2 ∧ ¬γ3)}
4  loseBuechi = {(s0, ¬γ1 ∨ γ1 ∧ ¬γ2 ∧ ¬γ3 ∨ γ1 ∧ γ2 ∧ γ3),
5                (s1, false)}
6  ReqG[s0] = {
7      ({g1}, γ1 ∧ γ2 ∧ ¬γ3),
8      ({g2}, γ1 ∧ ¬γ2 ∧ γ3)}
9
10 ReqG[s1] = {
11     ({g1}, γ2 ∧ ¬γ3),
12     ({g2}, ¬γ2 ∧ γ3)}
```

Eine Transition ist noch auf dem *Waiting*-Stack verblieben. Es handelt sich hierbei um die Transition e_2 mit feature expression $\neg\gamma_2 \wedge \gamma_3$. Da (g_1) ein noch nicht besuchter goal state ist, wird dieser der *Goal*-Map hinzugefügt und es erfolgt zweimal eine Rückpropagierung. Zuerst für (s_1), dann wieder für (s_0). Danach ist der erste Durchlauf von *FOTFR* abgeschlossen und die globalen Variablen sind wie folgt belegt:

```
1  Goals = {g1, g2}
2  WinR = {(s0, γ1 ∧ ¬γ2 ∧ γ3 ∨ γ1 ∧ γ2 ∧ ¬γ3),
3          (s1, ¬γ2 ∧ γ3 ∨ ∧γ2 ∧ ¬γ3)}
4  loseBuechi = {(s0, ¬γ1 ∨ γ1 ∧ ¬γ2 ∧ ¬γ3 ∨ γ1 ∧ γ2 ∧ γ3),
5                 (s1, ¬γ2 ∧ ¬γ3 ∨ ∧γ2 ∧ γ3)}
6  ReqG[s0] = {
7      ({g1}, γ1 ∧ γ2 ∧ ¬γ3),
8      ({g2}, γ1 ∧ ¬γ2 ∧ γ3)}
9
10 ReqG[s1] = {
11     ({g1}, γ2 ∧ ¬γ3),
12     ({g2}, ¬γ2 ∧ γ3)}
```

Als nächstes wird geschaut, ob für die gefundenen goal states gesagt werden kann, ob diese *winning* oder *losing* sind (Zeile 8, Algorithmus 1). Die erste Prüfung für den goal state (g_1) sieht wie folgt aus:

$$\neg(\gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3 \Rightarrow false \vee false)$$

$$\equiv \gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3$$

Somit wird für diesen goal state $FOTFR(g_1,\ \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1\gamma_2 \wedge \neg\gamma_3)$ aufgerufen (Zeile 10). Hier wird als erstes der noch nicht explorierte Zustand (g_1) mit der Transition e_3 untersucht. Abbildung 3.4 veranschaulicht das.

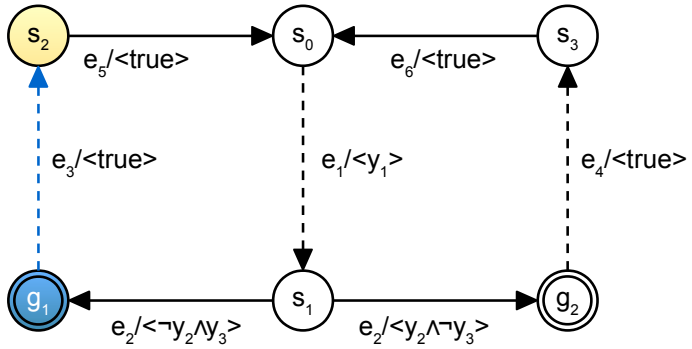


Abbildung 3.4: Synthese Algorithmus Beispiel 1 - Schritt 3

Nachdem die Transition e_3 wieder auf den *Waiting*-Stack gelegt wurde (Zeile 21, Algorithmus 2), wird die Funktion $ProcessUncontrollableTransitions(g_1)$ aufgerufen. Die globalen Variablen sind danach wie folgt belegt:

```
1  Goals = {g1, g2}
2  WinR = {(s2, false),
3          (s0, γ1 ∧ ¬γ2 ∧ γ3 ∨ γ1 ∧ γ2 ∧ ¬γ3),
4          (s1, ¬γ2 ∧ γ3 ∨ ∧γ2 ∧ ¬γ3)}
5  loseBuechi = {(s0, ¬γ1 ∨ γ1 ∧ ¬γ2 ∧ ¬γ3 ∨ γ1 ∧ γ2 ∧ γ3),
6                 (g1, true),
7                 (s1, ¬γ2 ∧ ¬γ3 ∨ ∧γ2 ∧ γ3)}
8  ReqG[s0] = {
9      ({g1}, γ1 ∧ γ2 ∧ ¬γ3),
10     ({g2}, γ1 ∧ ¬γ2 ∧ γ3)}
11
12 ReqG[s1] = {
13     ({g1}, γ2 ∧ ¬γ3),
14     ({g2}, ¬γ2 ∧ γ3)}
```

Eine Rückpropagierung erfolgt nicht, da FOTFR für (g_1) aufgerufen wurde und der Eintrag in *Depend*-Map auf die leere Menge zeigt. Die *while*-Bedingung (Zeile 5) evaluiert zu *false*:

$$\neg(\gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3 \;\Rightarrow\; false \;\vee\; true)$$
$$\equiv\; false$$

Somit ist (g_1) nicht mehr unentschieden und *FOTFR* wird beendet. Der nächste unentschiedene goal state ist (g_2). Also wird für diesen $FOTFR(g_2, \gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3 \vee \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3)$ aufgerufen.

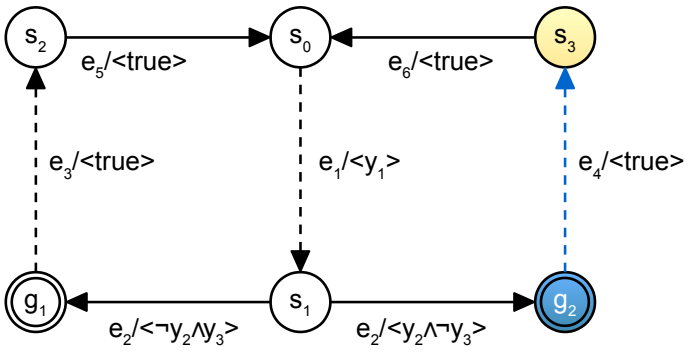


Abbildung 3.5: Synthese Algorithmus Beispiel 1 - Schritt 4

Wie in Abbildung 3.5 zu sehen, wird die Transition e_4 untersucht. Hier ist der

Durchlauf des Algorithmus analog zur Transition e_3. Die globalen Variablen sehen nach Rückpropagierung wie folgt aus:

```
1  Goals = {g1, g2}
2  WinR = {(s2, false),
3          (s0, γ1 ∧ ¬γ2 ∧ γ3 ∨ γ1 ∧ γ2 ∧ ¬γ3),
4          (s3, false),
5          (s1, ¬γ2 ∧ γ3 ∨ ∧γ2 ∧ ¬γ3)}
6  loseBuechi = {(s0, ¬γ1 ∨ γ1 ∧ ¬γ2 ∧ ¬γ3 ∨ γ1 ∧ γ2 ∧ γ3),
7                 (g1, true),
8                 (s1, ¬γ2 ∧ ¬γ3 ∨ ∧γ2 ∧ γ3),
9                 (g2, true)}
10 ReqG[s0] = {
11     ({g1}, γ1 ∧ γ2 ∧ ¬γ3),
12     ({g2}, γ1 ∧ ¬γ2 ∧ γ3)}
13
14 ReqG[s1] = {
15     ({g1}, γ2 ∧ ¬γ3),
16     ({g2}, ¬γ2 ∧ γ3)}
```

Nun ist auch (g_1) nicht mehr unentschieden und die *while*-Bedingung in Zeile 8 des Algorithmus 1 ist für beide goal states *false*. Somit terminiert der Algorithmus.

Anhand dieses Beispiels wurde für den Synthesealgorithmus gezeigt, dass dieser aus den beiden validen Produkten

$$\rho_1 \equiv \gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3$$

$$\rho_2 \equiv \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3$$

die beiden realisierbaren Produkte

$$\varphi_1 \equiv \gamma_1 \wedge \neg\gamma_2 \wedge \gamma_3$$

$$\varphi_2 \equiv \gamma_1 \wedge \gamma_2 \wedge \neg\gamma_3$$

korrekt ableitet. In diesem Fall sind valide und realisierbare Produkte identisch.

3.4 Gegenbeispiel

In diesem Abschnitt soll ein Gegenbeispiel für den Algorithmus erbracht werden, in dem der Algorithmus aus einer Menge valider Produkte, keine realisierbaren Produkte findet.

Der Algorithmus wird mit $FOTFB(s_0, \gamma_1)$ aufgerufen. Es erfolgt der erste Reachability-Durchlauf für (s_0). Dazu wird als erstes die Transition e_1 vom *Waiting*-Stack geholt wie in Abbildung 3.6 zu sehen.

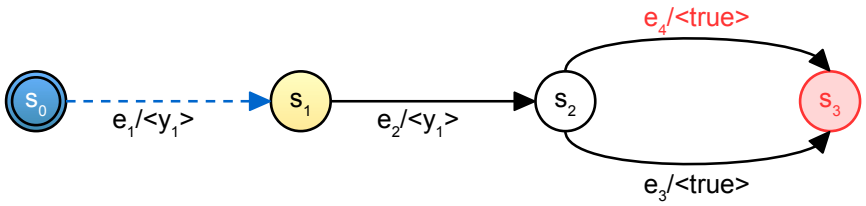


Abbildung 3.6: Synthese Algorithmus Beispiel 2 - Schritt 1

Der Zustand (s_1) ist noch nicht als besucht markiert und auch kein goal state. Es wird die nachfolgende Transition e_2 auf den Stack gelegt und im Anschluss untersucht. Abbildung 3.6 veranschaulicht das.

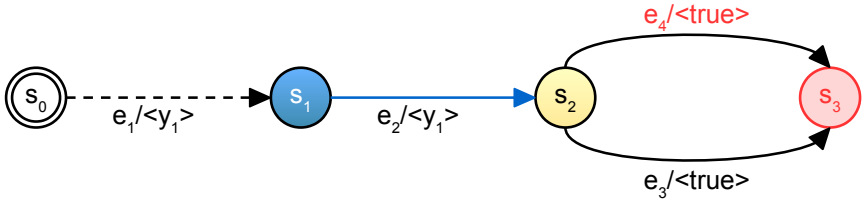


Abbildung 3.7: Synthese Algorithmus Beispiel 2 - Schritt 2

Der Zielzustand (s_2) ist ebenfalls noch nicht besucht und auch kein goal state. Seine beiden ausgehenden Transitionen e_3 und e_4 werden auf den *Waiting*-Stack gelegt. Anschließend wird e_4 untersucht (Abbildung 3.8). Der Zielzustand (s_3) - hier rot markiert - löst eine *Safety Violation* aus.

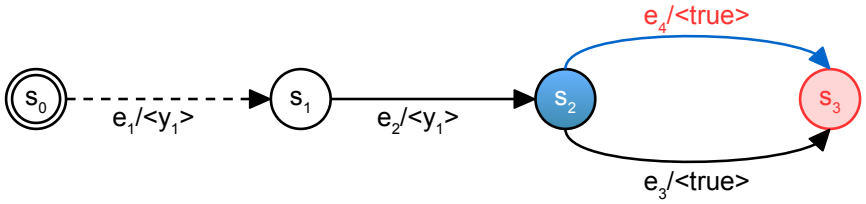


Abbildung 3.8: Synthese Algorithmus Beispiel 2 - Schritt 3

Daher wird wie in Abbildung 3.9 zu sehen, die Transition e_3 vom *Waiting*-Stack geholt. Diese Transition führt ebenfalls zu (s_3). Dem Zustand mit der Safety-Violation. Dieser ist bereits als besucht markiert und es erfolgt eine Rückwärtsreevaluierung.
Die globalen Variablen sind anschließend wie folgt belegt:

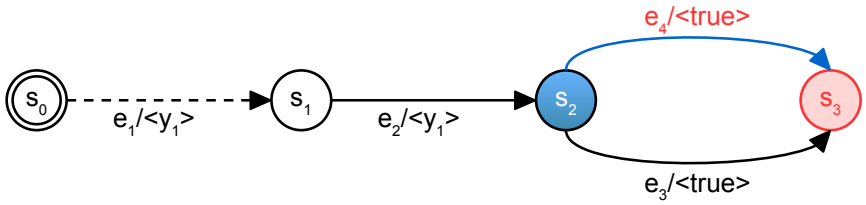


Abbildung 3.9: Synthese Algorithmus Beispiel 2 - Schritt 4

```
Goals = ∅
WinR = {(s1, false),
        (s2, false),
        (s3, false)}
loseBuechi = {(s2, true)}
ReqG = ∅
```

Nun erfolgt zweimal eine Rückpropagierung. Die Belegung der globalen Variablen sieht danach wie folgt aus:

```
Goals = ∅
WinR = {(s1, false),
        (s2, false),
        (s3, false)}
loseBuechi = {(s1, true),
              (s2, true)}
ReqG = ∅
```

Der erste Aufruf von *FOTFR* ist beendet. Da aber keine goal states gefunden wurden, die *Goals*-Map ist leer, sind keine unentschiedenen goal states vorhanden. Der Algorithmus terminiert und gibt aus, dass keine realisierbaren Produkte gefunden wurden aus der Menge $\{\rho_1 \equiv \gamma_1\}$ der validen Produkte.

Kapitel 4

Implementation

Die Implementation wurde mit Hilfe von zwei Plug-In-Projekten in Eclipse realisiert. Abschnitt 4.1 befasst sich mit der Architektur der Projekte. In 4.2 wird die Ein- und Ausgabe der Erweiterung erläutert und in Abschnitt 4.3 werden benötigte Komponenten von Drittanbietern beschrieben.

4.1 Projektarchitektur

org.scenariotools.productline.synthesis.fotfb.ui

Dieses Plug-In-Projekt besteht aus einer Kontextmenüerweiterung, wodurch die OTF-Synthese ausgeführt wird. Der Benutzer kann mit einem Rechtsklick auf eine ScenarioRunConfiguration-Datei die Synthese auswählen, wie in Abbildung 4.1 zu sehen.

Hierbei wird dann ein Hintergrundprozess angestoßen, der die FOTF-Synthese startet.

org.scenariotools.productline.synthesis.fotfb

Das zweite Projekt ist die eigentliche Implementation des Synthesealgorithmus. Dieser wird wie oben erwähnt über die Kontextmenüerweiterung aufgerufen und zeigt dem Benutzer im Anschluss ein Meldungsfenster wie in Abschnitt 4.2 beschrieben.

4.2 Ein- und Ausgabe

Die Ein- und Ausgaben der FOTF-Synthese werden in Abbildung 4.2 veranschaulicht.

Das Plug-In erhält wie oben erwähnt eine ScenarioRunConfiguration-Datei als Eingabe. Dort sind alle relevanten Daten für die Simulation der zu prüfenden Produktlinienspezifikation hinterlegt, wie die MSD-Spezifikation

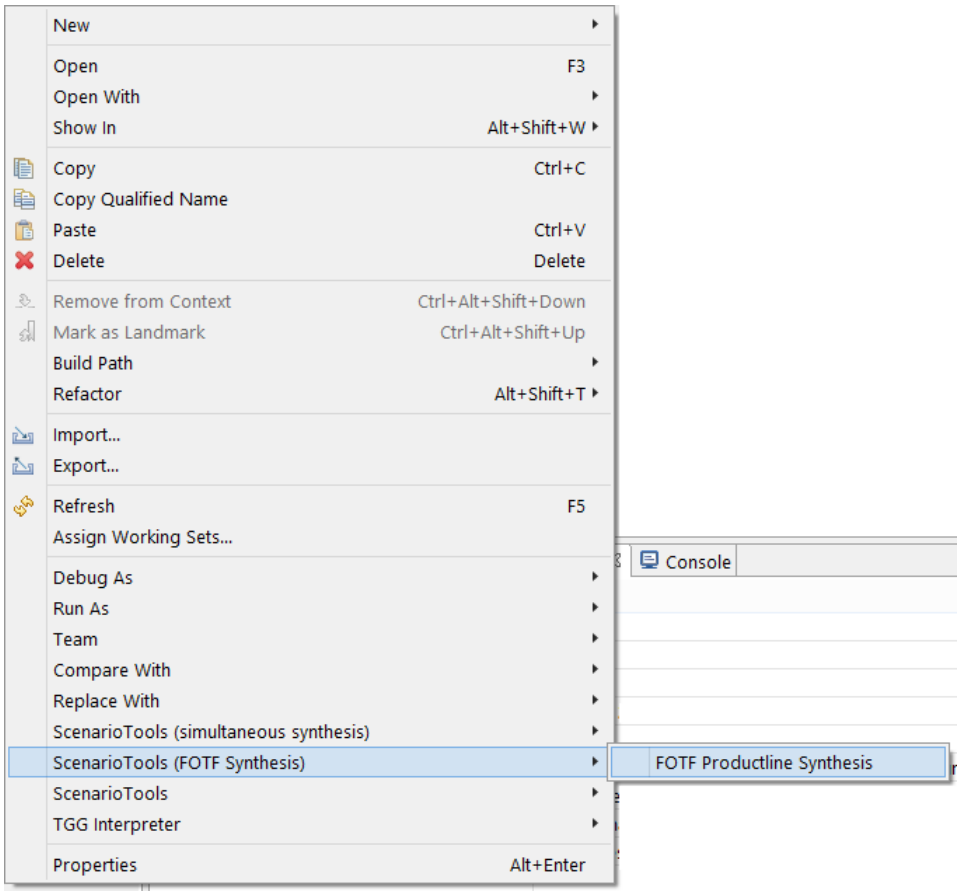


Abbildung 4.1: Kontextmenüerweiterung des Synthesealgorithmus

und das Feature Diagram. Daraus kann dann der FGG aufgebaut werden, auf dem die FOTF-Synthese läuft.

Nach Abschluss der Synthese wird dem Benutzer ein Meldungsfenster angezeigt. Abbildung 4.3 zeigt ein solches Meldungsfenster.

Dieses enthält Informationen darüber, ob es eine Büchi-Strategie gibt. Also ob mindestens ein Produkt realisierbar ist. Es werden die Anzahl der gefundenen goal states und explorierten Zustände samt Transitionen angezeigt. Die benötigte Zeit in Millisekunden wird ebenfalls angezeigt sowie die Anzahl der validen und realisierbaren Produkte, einschließlich der KNF ihrer feature expressions.

Mit Hilfe der MsdRuntime-Datei kann der FGG generiert werden. ScenarioTools bietet hier die Möglichkeit, diesen in das PDF-Format zu exportieren.

Abbildung 4.2: Ein- und Ausgaben der FOTF-Synthese

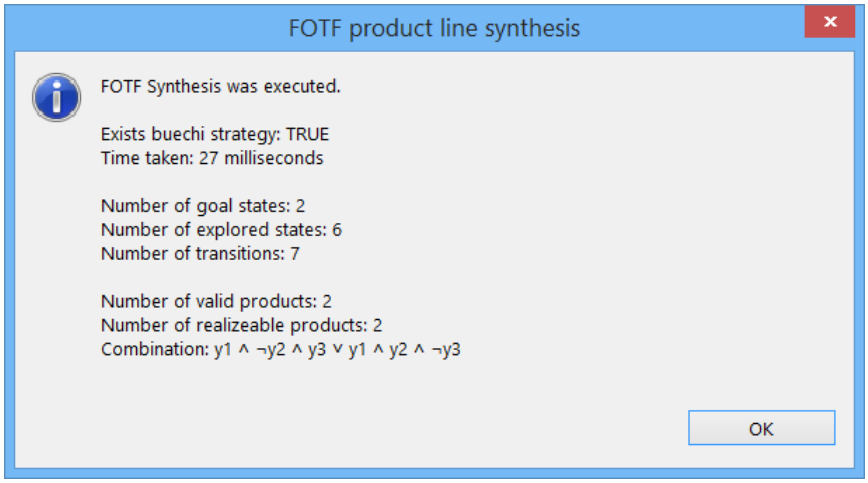


Abbildung 4.3: Ausgabefenster der FOTF-Synthese

4.3 Abhängigkeiten von Drittanbietern

JDD, a pure Java BDD and Z-BDD library

Der Synthesealgorithmus benutzt die JDD-Bibliothek[1]. Das ist eine Bibliothek für die Nutzung von binären Entscheidungsdiagrammen. Sogenannte Binary Decision Diagrams (BDDs) - hier als binärer Entscheidungsbaum implementiert - kommen häufig beim Model Checking, zur formalen Verifikation oder auch bei Schaltkreisoptimierungen zum Einsatz. Bei der FOTF-Synthese werden BDDs für die feature expressions genutzt. Somit können diese effizient berechnet und dargestellt werden. Jedes feature des FDs bekommt hierfür ein eigene BDD Variable. Die JDD-Bibliothek stellt eine Menge an logischen Operationen zur Verfügung, die hier genutzt werden.

[1] `http://javaddlib.sourceforge.net/jdd`

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Da wir im Alltag immer stärker von Software unterstützt werden und viele Aufgaben von software-intensiven Systemen, gerade auch in kritischen Bereichen übernommen werden, ist es wichtig, schon bei deren Entwicklung genau darauf zu achten, dass keine Anforderungen verletzt werden und die Systeme das tun, was von ihnen verlangt wird. Erschwert wird die Entwicklung zusätzlich bei Produktlinien, da hier die verschieden Kombinationen an Features leicht exponentiell wachsen können, was die Prüfung jedes einzelnen Produkts erschwert. Zudem kann es bei einigen Feature-Kombinationen zu Widersprüchen kommen. Dafür bietet der szenariobasierte Entwurf eine gute Möglichkeit, das Gesamtverhalten durch Szenarien zu beschreiben. Dadurch ist der Entwickler in der Lage präzise zu sagen, was ein System als Reaktion auf bestimmte Ereignisse tun kann, tun muss oder nicht tun darf.

Solche szenariobasierten Spezifikationen können in ScenarioTools, einer Eclipse-basierten Werkzeugumgebung, modelliert, simuliert und ausgeführt werden. Mit Hilfe des Synthesealgorithmus ist es möglich, Widersprüche in der Spezifikation zu finden und für Produktlinienspezifikationen die Menge von realisierbaren Produkten von der Menge der validen Produkte zu trennen.

Das Ziel dieser Arbeit war es, den Synthesealgorithmus von Gressi [10] zu optimieren. Dazu wurde versucht, den Algorithmus on-the-fly zu optimieren und in ScenarioTools zu integrieren. Hierzu wurden zwei Plug-In-Projekte angelegt. Zum einen eine Kontextmenüerweiterung, die beim Rechtsklick auf eine ScenarioRunConfiguration-Datei die Synthese startet. Zum anderen die Implementation des OTF-Synthesealgorithmus.

Der Algorithmus wurde erfolgreich, prototypisch implementiert. Allerdings ist der Algorithmus noch nicht ganz vollständig. Es kann vorkommen,

dass die Synthese vorzeitig beendet wird, so dass nur eine Teilmenge der realisierbaren Produkte gefunden wird. Darüber hinaus findet der Algorithmus in einigen Fällen keine realisierbaren Produkte, obwohl er in einem weiteren Durchlauf derselben Spezifikation schließlich doch die korrekte Menge an realisierbaren Produkten findet. Dies liegt u.a. an der Tatsache, dass der Play-Out-Algorithmus nichtdeterministisch arbeitet, so dass der FGG unterschiedlich aufgebaut wird.

5.2 Ausblick

Damit der Algorithmus vollständig wird, ist eine nähere Untersuchung der derzeitigen Rückpropagierung erforderlich. Außerdem ist ein Performanzvergleich mit der featured Synthese von Gressi [10] interessant, um zu schauen, unter welchen Bedingungen die on-the-fly Synthese möglicherweise performanter ist.

Eine weitere Möglichkeit einer Optimierung wäre die parallele Exploration eines FG-Graphen. Hierzu könnte für jeden Zweig ein zusätzlicher Thread genutzt werden, der diesen Teilbaum exploriert. Anschließend werden die Ergebnisse von jedem Thread synchronisiert und es erfolgt eine Rückpropagierung entlang des Hauptastes des FG-Graphen. Für sehr große und komplexe Spezifikationen könnte sich hierdurch ein zusätzlicher Geschwindigkeitsvorteil ergeben, der aber bei kleineren Spezifikationen wohl eher gering oder möglicherweise gar nicht zum Ausdruck kommen könnte, auf Grund der zusätzlichen Thread-Synchronisation und dem damit verbundenem Verwaltungsaufwand.

Literaturverzeichnis

[1] F. Alizon, S. B. Shooter, and T. W. Simpson. Henry Ford and the Model T: lessons for product platforming and mass customization. *Design Studies*, 30:588–605, 2009.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey. Product line practice workshop report. Technical report, Carnegie Mellon University, June 1997.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sep 1991.

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

[6] M. Cordy, J. Greenyer, P. Heymans, and A. Molzam Sharifloo. Efficient consistency checking of scenario-based product line specifications. In M. P. E. Heimdahl and P. Sawyer, editors, *Proceedings of the 20th International Requirements Engineering Conference (RE 2012)*, pages 161–170. IEEE, 2012.

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

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

[9] J. Greenyer, A. Molzam Sharifloo, M. Cordy, and P. Heymans. Features meet scenarios: modeling and consistency-checking scenario-based product line specifications. *Requirements Engineering*, 18(2):175–198, 2013.

[10] E. Gressi. Featured synthesis of scenario-based software product line specifications. Master's thesis, POLITECNICO DI MILANO, 2012-2013.

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

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

[13] D. Harel and I. Segall. Synthesis from scenario-based specifications. *Journal of Computer and System Sciences*, 78(3):970 – 980, 2012. In Commemoration of Amir Pnueli.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie Mellon University, November 1990.

[15] P. C. C. Linda M. Northrop. A framework for software product line practice. `http://www.sei.cmu.edu/productlines/frame_report/introduction.htm`, 2012. Letzter Zugriff März 2015.

[16] P. C. C. Linda M. Northrop. A framework for software product line practice. `http://www.sei.cmu.edu/productlines/frame_report/what.is.a.PL.htm`, 2012. Letzter Zugriff März 2015.

[17] L. ren Jen and Y. jye Lee. Working group. ieee recommended practice for architectural description of software-intensive systems. *IEEE Architecture*, pages 1471–2000, 2000.

[18] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, RE '06, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society.

[19] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, Jan. 1997.

Abbildungsverzeichnis

List of Algorithms