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

Simulation und Analyse von probabilistischen szenariobasierten Spezifikationen

Simulation and Analysis of Probabilistic Scenario-based Specifications

Bachelorarbeit

im Studiengang Informatik

von

Dennis Griethe

Prüfer: Prof. Dr. Greenyer
Zweitprüfer: Prof. Dr. Schneider
Betreuer: Dipl.-Inf. Daniel Gritzner

Hannover, 18.04.2018

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 18.04.2018

_____

Dennis Griethe

Zusammenfassung

Mit der ständig zunehmenden Komplexität reaktiver Systeme wird auch das Erstellen einer Anforderungsspezifikation immer schwieriger. Um den Prozess hin zu einer vollständigen, konsistenten Spezifikation zu erleichtern, hat das Fachgebiet Software Engineering der Leibniz Universität Hannover ScenarioTools entwickelt. Damit lassen sich szenariobasierte Spezifikation intuitiv erstellen und auf Konsistenz überprüfen. ScenarioTools beinhaltet allerdings noch keine Möglichkeit, Hard- oder Softwarefehler, wie nicht ankommende Bus-Nachrichten oder fehlerhafte Sensoren zu modellieren.
In dieser Arbeit diskutieren wir daher eine Variante, probabilistisches Verhalten in der Scenario Modeling Language zu modellieren und anschließend grundlegende Reliability-Fragen zu beantworten. Dazu stellen wir eine neu entwickelte Kontrollstruktur in SML vor.

Abstract

<Englischer Titel der Arbeit>

A short summary of the thesis in about 200 words.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Struktur der Arbeit 2

2 Grundlagen **3**
2.1 Scenario Tools 3
2.1.1 Klassenmodelle 3
2.1.2 Modellieren in SML 3
2.1.3 Simulation der Spezifikation 6
2.2 Prism Model Checker 6
2.2.1 Modellieren in Prism 7
2.2.2 Simulieren in Prism 9

3 Ziele dieser Arbeit **12**
3.1 Das HVCS 12
3.1.1 Die Strategie 15
3.2 Reliability Fragen 16
3.2.1 Reliability 16
3.2.2 Availability 17
3.2.3 Maintainability 17
3.2.4 Safety 17
3.2.5 Einordnung in diese Arbeit 17

4 Ausarbeitung der ScenarioTools Erweiterung **18**
4.1 Erweiterungen in SML 18
4.1.1 Probabilistic Alternatives im HVCS 20
4.2 Anpassen der ScenarioTools Simulationsumgebung 21
4.3 Controller Synthese mit PA 23
4.4 Prism Modelle generieren 24

5 Modellanalyse mit Prism Properties **27**
5.1 Beispielanwendungen 27
5.1.1 Reliability 27
5.1.2 Safety 30
5.2 Properties aus ScenarioTools 30

5.2.1 Properties in SML 30

6 Diskussion **32**
6.1 Bewertung des Lösungsansatzes 32
6.2 Mehraufwand der PA 33

7 Verwandte Arbeiten **34**
7.1 Alternativen zu ScenarioTools 34
7.2 Alternativen zu Prism 34
7.3 Zusammenfassung 34
7.4 Ausblick 35

A Anhang **36**
A.1 Inhalte der CD 36
A.2 Strategien 36
A.3 Code 38

Kapitel 1

Einleitung

Cyberphysische Systeme sind in der modernen Softwareentwicklung von stetig zunehmender Bedeutung. Zu solchen Systemen zählen zum Beispiel Industrieroboter, Anlagen zur Verkehrsteuerung und Stromnetzsteuerungstechnik. Ein einwandfreies Zusammenspiel von Soft- und Hardwarekomponenten ist also unabdingbar, um den reibungslosen Ablauf eines solchen Systems zu gewährleisten.
An erster Stelle der Entwicklung eines solchen reaktiven Systems steht die Erhebung der Anforderungen. Dabei arbeiten domänenkundige Fachkräfte eng mit einem Requirements Engineer zusammen, um zu ermitteln, wie sich Effizienz und Sicherheit zu einem möglichst performanten Endprodukt vereinen lassen. Dieser überaus wichtige Schritt auf dem Weg zum Deployment des fertigen Systems lässt sich bei immer umfangreicheren Systemen, mit immer mehr Komponenten, zunehmend schwieriger manuell überblicken. Um das erstellen einer konsistenten, vollständigen Spezifaktion zu erleichtern, hat das Fachgebiet Software Engineering der Leibniz Universität Hannover *ScenarioTools*[1] entwickelt. Dabei handelt es sich um eine Sammlung von Eclipse[2] Plugins, die eine Entwicklungsumgebung zum Erstellen szenariobasierter Spezifikationen bildet. Mit Hilfe der *Scenario Modelling Language*, einer domänenspezifischen Sprache, kann man intuitiv Spezifikationen erstellen, validieren und testen. Außerdem gibt es die Möglichkeit, sich alle möglichen Systemzustände als Graph ausgeben zu lassen oder eine Strategie zu erstellen, welche dann auch zur Generierung von ausführbarem Code genutzt werden kann.
ScenarioTools beinhaltet allerdings noch keine Möglichkeit Hard- oder Softwarefehler, z.B. nicht ankommende BUS-Nachrichten, fehlerhafte Sensoren oder verschlissene Teile, in die Spezifikation mit einzubeziehen. Dadurch ist es dem Ingenieur nicht möglich, zuverlässigkeitsrelevante Fragen zu beantworten. Dabei sind diese, aus dem Reliability Engineering stammenden,

[1] `www.scenariotools.org`
[2] `www.eclipse.org`

Fragestellungen gerade im Bezug auf Effizienz und Sicherheit nicht zu vernachlässigen. In einer Produktionsanlage ist eine Abschätzung, nach wie vielen produzierten Stücken die Anlage gewartet werden sollte, hilfreich. In sicherheitsrelevanten Anwendungen, wie der Verkehrsleitung, ist es von Interesse, wie zuverlässig die einzelnen Systemkomponenten sein müssen, um einen gewissen Sicherheitsstandard einzuhalten. Um diese und weitere Fragestellungen des Reliability Engineering zu beantworten, muss die Möglichkeit geschaffen werden Fehlerwahrscheinlichkeiten und damit assoziierte Nachteile nachvollziehbar darzustellen und untersuchen zu können.
In dieser Arbeit erläutern und diskutieren wir deshalb eine Möglichkeit, SML dahingehend zu erweitern, dass Wahrscheinlichkeiten und damit zusammenhängende Veränderungen des Zustandsraums des Systems intuitiv modelliert und evaluiert werden können. Dabei erweitern wir SML um die nötige Funktionalität und diskutieren die Darstellung in *ScenarioTools* Ein wichtiger Bestandteil dieser Arbeit ist die Verwendung des *Prism Model Checkers*[3]. Ein eigenständiges Programm, dass probabilistische Auswertungen und Simulationen für verschiedene stochastische Modelle ermöglicht. Die Verwendung von Prism erspart die eigenen Implementierung einer stochastischen Auswertung in *ScenarioTools*. Stattdessen wird die fertige Strategie nach Prism konvertiert und dort ausgewertet. Gleiches geschieht mit den Zuverlässigkeitsfragen, welche, möglichst intuitiv, in *ScenarioTools* gestellt und dann automatisch nach Prism konvertiert werden können.
Diese Arbeit erstellt das dafür notwendige Konzept und schlägt Änderungen an SML vor. Des Weiteren diskutieren wir die zu tätigenden Änderungen an *ScenarioTools*.

1.1 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert. In Kapitel 1 wir die Problemstellung, der Lösungsansatz und die Struktur der Arbeit beschrieben. In Kapitel 2 werden die Grundlagen, auf denen diese Arbeit basiert, erläutert. Dabei handelt es sich hauptsächlich um *ScenarioTools* und *Prism* In Kapitel 3 werden die zu erreichenden Ziele dieser Arbeit genauer beschrieben. In Kapitel 4 wird die Erweiterung von SML zur Darstellung probabilistischer Szenarien detailliert vorgestellt. In Kapitel 5 wird erläutert, wie Reliability Metriken aus *ScenarioTools* nach Prism konvertiert werden. In Kapitel 6 wird diskutiert, wie aufwendig eine Implementierung des vorgestellten Konzepts wäre und wie gut die Ziele aus Kapitel 3 erreicht wurden. In Kapitel **??** werden Alternativen zu *ScenarioTools* und *Prism* vorgestellt, sowie alternative Herangehensweisen an das Problem konfrontiert. In Kapitel 7.2 werden die Erkenntnisse dieser Arbeit noch einmal zusammengefasst.

[3] `www.prismmodelchecker.org`

Kapitel 2

Grundlagen

In diesem Kapitel erlangen wir einen Einblick in die Funktionsweisen der beiden maßgebenden Programme, die wir benutzen. *ScenarioTools* und *Prism Model Checker*.

2.1 Scenario Tools

ScenarioTools ist eine Sammlung von Eclipse Plugins, die das Modellieren komplexer reaktiver Systeme erleichtert[3]. Darin werden verschiedene Objekte in Spezifikationen zusammengefasst und ihr Verhalten untereinander in Szenarien modelliert. Eine vollständige Spezifikation besteht aus vier Teilen: Einem **Klassenmodell** und daraus instanziiertem **Objektmodell** 2.1.1, einer Spezifikation in **SML** 2.1.2, sowie einer **Runconfiguration**, welche das Objektmodell und die SML Spezifikation verbindet.

2.1.1 Klassenmodelle

An erster Stelle des Entwicklungsprozesses mit *ScenarioTools* steht die Erstellung eines Klassenmodells, das die Domäne beschreibt. Dafür werden die einzelnen Systemkomponenten in *EMF ecore* definiert. Jede Klasse kann dabei Methoden, Attribute und Variablen beinhalten. Dieses Klassenmodell wir später in eine Spezifikation importiert, sodass dann mittels SML auf die Eigenschaften jeder Klasse zugegriffen werden kann. Zur Simulation der Spezifikation wird später noch ein Objektmodell benötigt. In diesem werden spezifische Objekte den im Klassenmodell definierten Klassen zugeordnet. Bei diesen Objekten handelt es sich um Instanzen der Klassen aus dem Klassendiagramm.

2.1.2 Modellieren in SML

Die *Scenario Modelling Language*, kurz SML, ist einer der Grundsteine von *ScenarioTools*. Es handelt sich dabei um eine *domain specific language*,

```
1   import "../../model/highvoltagecoupling.ecore"
2
3   specification HighvoltagecouplingSpecification {
4
5   domain highvoltagecoupling
6
7   controllable {
8   Controller
9   }
10
11  collaboration PlugAndStart {
12
13  static role Socket socket
14  static role Relays relays
15  static role StartButton startButton
16  static role Controller controller
17
18
19  /*
20  * When the plug is plugged into the socket,
21  * then the socket must be locked.
22  */
23  guarantee scenario WhenPluggedThenLock{
24  socket->controller.plugged()
25  strict urgent controller->socket.lock()
26  }
27
28  assumption scenario NoUnplugBetweenLockAndUnlock{
29  controller->socket.lock()
30  controller->socket.unlock()
31  }constraints[
32  forbidden socket->controller.unplugged()
33  ]
34  [...other Scenarios]
```

Code 1: Auszug aus der HVCS Spezifikation

mit der Spezifikationen auf Basis der genannten Klassenmodelle intuitiv entwickelt werden können. Eine Grundidee bei der Arbeit mit SML ist dabei die Unterscheidung zwischen kontrollierbaren und unkontrollierbaren Komponenten einer Spezifikation, welche als Klassen modelliert werden. Die unkontrollierbaren, oder auch Umwelt-gesteuerten, Klassen stellen dabei Ereignisse außerhalb der Kontrolle des zu entwerfenden Software-Controllers dar. Dabei kann es sich zum Beispiel um die Sensoren eines Autos, oder Aktionen eines anderen Systems handeln. Die kontrollierbaren Klassen stellen Komponenten des Systems dar, welche es zu entwickeln gilt, wie z.B die Steuerung eines Industrieroboters.

Abbildung 1 zeigt einen Auszug aus dem SML Code einer *High Voltage Coupling System* Spezifikation. Dabei geht es darum, eine Steckerverbindung gefahrlos schließen und öffnen zu können. Eine Spezifikation besteht, im wesentlichen, aus der Einordnung der Klassen und einer unbestimmten Anzahl an *Collaborations*. In diesem Beispiel ist unsere einzige kontrollierbare Klasse der *Controller*. In einer *Collaboration* werden Rollen definiert, die den Objekten des Objektmodells und damit den Systemkomponenten entsprechen. Diese Definition kann entweder **statisch** sein, sodass immer nur ein Objekt diese Rolle annehmen kann, oder **dynamisch**. Bei einer dynamischen Rollenverteilung können mehrere Objekte des Objektmodells diese Rolle

annehmen. Unabhängig davon ist eine Rolle immer über eine Klasse aus dem Klassendiagramm getypt und kann daher nur von einem Objekt dieser Klasse angenommen werden. Außerdem enthält eine *Collaboration* Szenarien in denen das Verhalten des Systems definiert wird. Es gibt zwei Arten von Szenarien: *guarantee* und *assumption scenarios*. In einem *guarantee scneario* wird beschrieben, wie das System auf bestimmte Ereignisse reagiert, während *assumption scenarios* beschreiben, was in der Umwelt passieren kann und was nicht. Beides wird durch eine Abfolge von *messages* modelliert. Diese stellen Nachrichten von einer Komponente an eine andere dar. Im Szenario *WhenPluggedThenLock* meldet zum Beispiel das *socket* an den *controller*, mittels der Methode *plugged*, dass ein Stecker eingesteckt wurde. Tritt diese Nachricht nun auf, ist das Szenario aktiv und die zweite Nachricht wird erwartet. SML bietet, durch eine Vielzahl an Schlüsselwörtern, Möglichkeiten, *Safety* und *Liveness* conditions zu modellieren. Mittels Liveness Conditions können wir dem Controller vorschreiben, dass bestimmte Nachrichten innerhalb definierter Zeitfenster aktiviert werden müssen. Safety Conditions werden genutzt, um die Sicherheit des Systems zu gewährleisten. So wird im Beispiel(Zeile 25) mit Hilfe des Schlüsselworts *strict* bestimmt, dass keine andere Nachricht des Szenarios gesendet werden darf, solange die strict Nachricht aktiv ist. Es darf also kein weiterer Stecker eingesteckt werden, bevor dieser verschlossen ist. In derselben Zeile wird mit *urgent* modelliert, wann die Nachricht gesendet werden muss. In diesem Fall dürfen nur andere urgent oder *committed* Nachrichten vorher gesendet werden.
Abseits dieser beiden Modellierungsaspekte bietet SML noch die Möglichkeiten üblicher Programmierkonstrukte, wie **Schleifen** und **Alternativen**. Schleifen heißen in SML *Loops* und funktionieren wie eine *while*-Schleife in anderen Hochsprachen. Eine Bedingung im Schleifenkopf bestimmt, wie oft der Schleifenkörper ausgeführt wird. Das folgende Beispiel zeigt einen unendlichen Loop, da die Bedingung immer wahr ist:

```
1    while[ true ]{
2        a->b.repeatMe()
3    }
```

Die Alternative ist die wichtigste Kontrollstruktur für diese Arbeit. Mit ihr wird in SML eine Verzweigung realisiert. Dabei teilt sich der Szenarioablauf in mehrere mögliche Fälle, genannt Cases. Welcher Case letztendlich gewählt wird, wird durch die Initialisierungs-Nachricht, die erste Nachricht jeden Cases, entschieden. Sobald sie auftritt, ist die Alternative entschieden und das Szenario fährt mit dem gewählten Case fort. Im folgenden Beispiel wird der erste Case gewählt wenn die Nachricht *a->b.chooseOne()* auftritt; der zweite, wenn *a->b.chooseTwo()* auftritt.

```
1    alternative{
2        a->b.chooseOne()
3    }or{
4        a->b.chooseTwo()
5    }
```

2.1.3 Simulation der Spezifikation

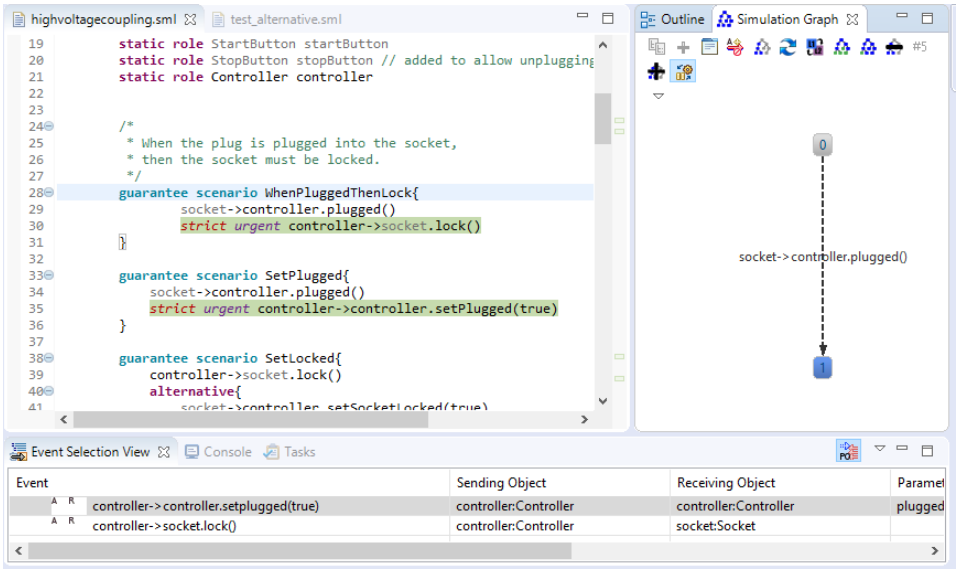

Abbildung 2.1: Die Simulationsperspektive von *ScenarioTools*

Zum Testen der geschriebenen Spezifikation steht einem in *ScenarioTools* eine Simulationsumgebung auf Basis der Eclipse Debug Umgebung zur Verfügung. Abbildung 2.1 zeigt deren Kernelemente. Nach dem Starten einer manuellen Simulation, sehr ähnlich dem genannten Debugging-Vorgang, werden im unteren Feld erwartete Nachrichten angezeigt, welche man dann auswählen und somit aktivieren kann. Die Folgen einer Nachricht werden dann sowohl rechts im Simulationsgraphen, wie auch links im Quellcode angezeigt. Links sieht man dabei, welche Nachrichten wo im Code jetzt erwartet werden, während der Simulationsgraph den bisher erkundeten Teil des Zustandsraumes des Systems darstellt. Alternativ zur manuellen Simulation kann man sich auch den Zustandsraum als Graphen ausgeben lassen.
Außerdem bietet *ScenarioTools* die Möglichkeit, die Spezifikation auf Realisierbarkeit zu prüfen. Dabei wird geprüft, ob die Umgebung das System durch ausnutzen der *assumptions* zwingen kann modellierte *safety-* oder *liveness conditions* zu verletzen. Sollte dabei eine Abfolge von Systemreaktionen gefunden werden, die zu keiner Verletzung führt, kann man sich diese *Strategie* ebenfalls als Graph ausgeben und anzeigen lassen.

2.2 Prism Model Checker

Prism Model Checker, in dieser Arbeit, zur besseren Lesbarkeit, *Prism* genannt, ist ein Programm zur Analyse probabilistischer Systeme[8]. Es kann

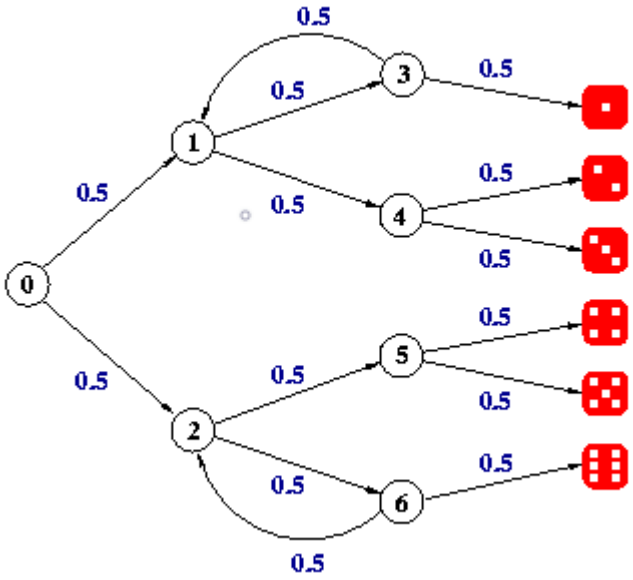

Abbildung 2.2:
Eine graphische Darstellung der Emulation eines Würfels durch Münzwürfe.[1]

mehrere stochastische Prozesse interpretieren. In dieser Arbeit werden wir uns allerdings nur einem genauer befassen. Der *Deterministic Time Markov Chain* (DTMC). Eine DTMC beschreibt die Wahrscheinlichkeit $P(s, s')$ von einem Zustand s einen anderen Zustand s' zu erreichen. Eine ihrer Besonderheiten ist dabei die Annahme, dass P dabei nur vom Zustand s und nicht von seinen Vorgängern abhängt. Prism benötigt als Eingabe ein, in einer probabilistischen Abwandlung von *Reactive Modules[2]* definiertes, Modell, sowie eine Menge von Eigenschaften, den *Prism Properties*.

2.2.1 Modellieren in Prism

Um einen kurzen Einblick in die Modellierung in Prism zu gewinnen, schauen wir uns ein Beispiel aus dem Prism Tutorial [1] an. Dabei wird der Wurf eines sechsseitigen Würfels mittels eines von *Knuth* und *Yao* entwickelten Algorithmus, durch Münzwürfe, simuliert[7].

Abbildung 2.2 zeigt eine graphische Darstellung der Zielemulation, welche direkt die verschiedenen Zustände definiert und rechts die emulierte Augenzahl anzeigt. In Code 2 sehen wir die Emulation als DTMC Modell in Prism. Prism Modelle bestehen aus einer Anzahl an Modulen, welche

[1] `www.prismmodelchecker.org/tutorial/die.php`

```
1   dtmc
2
3   module die
4
5   // local state
6   s : [0..7] init 0;
7   // value of the die
8   d : [0..6] init 0;
9
10  [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
11  [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
12  [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
13  [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
14  [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
15  [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
16  [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
17  [] s=7 -> (s'=7);
18
19  endmodule
```

Code 2: Die Würfelemulation als Prism Modell.

jeweils das Verhalten eines Prozesses beschreiben. Jedes Modul beinhaltet Variablen, welche den Zustand des Modells oder andere Werte halten können. Im Beispiel werden mit einer Variable s die Zustände der Emulation gehandhabt, während d die gewürfelte Augenanzahl angibt. Die Logik verbirgt sich in den *commands*, welche im Beispiel in den Zeilen 10-17 zu sehen sind. Ein command besteht immer aus einem *guard*, gefolgt von einem oder mehreren *updates*.

```
1   [action] guard -> P(update 1) : update 1 + P(update2) : update 2 +...;
```

Wird ein guard als wahr ausgewertet, werden die updates durchgeführt. Wir können dabei jedes update mit einer Wahrscheinlichkeit versehen, um diese in der Simulation später zu nutzen. In unserem Beispiel sehen wir, dass wir jeden Zustand einmal als guard benutzen und mittels der updates die Zustandsübergänge definieren. Wird der Folgezustand auf $s' = 7$ gesetzt, wird auch der Würfelwert bestimmt. In Zeile 10 sehen wir z.B, dass wenn wir uns im Zustand $s = 0$ befinden, wir mit der Wahrscheinlichkeit 0.5 in den Folgezustand $s' = 1$ übergehen, oder mit der gleichen Wahrscheinlichkeit in den Zustand $s' = 2$. Die Wahrscheinlichkeiten eines Commands müssen dabei immer zu Eins summieren. Zu guter Letzt können wir in den *//* zu Beginn jedes commands einen einfachen Kommentar oder Namen für die Aktion vergeben, der in der Simulation angezeigt wird. Diese können wiederum genutzt werden um einzelne Transitionen zu identifizieren und mit *rewards* zu versehen. Rewards erlauben quantitative Aussagen über z.B vergangen Zeit, indem jede Transition eine reward zugewiesen bekommt, oder über fehlgeschlagene Prozesse, indem spezifische Transitionen mit rewards versehen werden. Definiert werden sie in eigenen Blöcken, von denen es durchaus mehrere geben kann. Code 3 zeigt zwei Beispiele für Rewardstrukturen in Prism. Im ersten wird jedem Zustand die Reward Eins zugeordnet. Dadurch kann man später in Properties abschätzen, wie viel

```
1
2     //Jeder Zustand erhaelt Reward 1
3     rewards "Total Time"
4         true : 1;
5     endrewards
6     //Rewards nach Variablen
7     rewards
8         s=7 : 1;
9         d=6 : 6;
10    endrewards
11    //Rewards nach action
12    rewards
13        [] true : 1;
14        [action] true : 2;
15    endrewards
```

Code 3: Beispiele für Reward-Strukturen in Prism

Zeit in Form von Zuständen vergangen ist, bis ein anderes Ereignis eintritt. Die Rewards sind immer an den guard, also die Bedingung, links vom ':' geknüpft. Ist sie wahr, erhält der entsprechende Zustand die spezifizierte Reward. Im zweiten Beispiel wird dem Zustand $s = 7$ damit eine Reward von 1 zugeordnet. Alle Zustände die die Bedingung d=6 erfüllen erhalten eine Reward von 6. Die Rewardblöcke können auch benannt werden. So können sie bei der Auswertung identifiziert werden. Der dritte Beispielblock teilt die Rewards den Transitionen zu, welche mit dem entsprechenden Kommentar(action-label) gekennzeichnet sind.

2.2.2 Simulieren in Prism

Prism hat die Möglichkeit, das Modell von Hand zu Simulieren, indem man entweder jede Transition per Hand wählt, ähnlich wie die Nachrichtenauswahl in der *ScenarioTools* Simulationsumgebung, oder eine bestimmte Anzahl Schritte anhand der vergebenen Wahrscheinlichkeiten simulieren lässt.

In Abbildung 2.3 sehen wir die Simulationsumgebung von Prism. Darin ist das Beispielmodell bereits um zwei Transitionen fortgeschritten, sodass wir uns nun im Zustand $s = 3$ befinden und unser Würfelwert noch unverändert $d = 0$ beträgt. Mittig oben sehen wir die Auswahl zur *Manual Exploration*, in der wir den Folgezustand des Modells bestimmen können. Darin wird uns, von Links nach Rechts, der Name des modules, oder unser Kommentar, die Wahrscheinlichkeit für dieses Update und das update selbst angezeigt. Rechts erhalten wir zusätzliche Informationen zum Stand der Simulation. Etwa, ob wir uns im Initialzustand befinden, oder ein *deadlock* erreicht haben. Wir können die Simulation auch automatisch voranschreiten lassen, indem wir links im Fenster einen Anzahl Schritte auswählen und simulieren lassen. Dabei wird standardmäßig, unabhängig von der gewählten Schrittanzahl, gestoppt, sobald eine Schleife entdeckt wird.

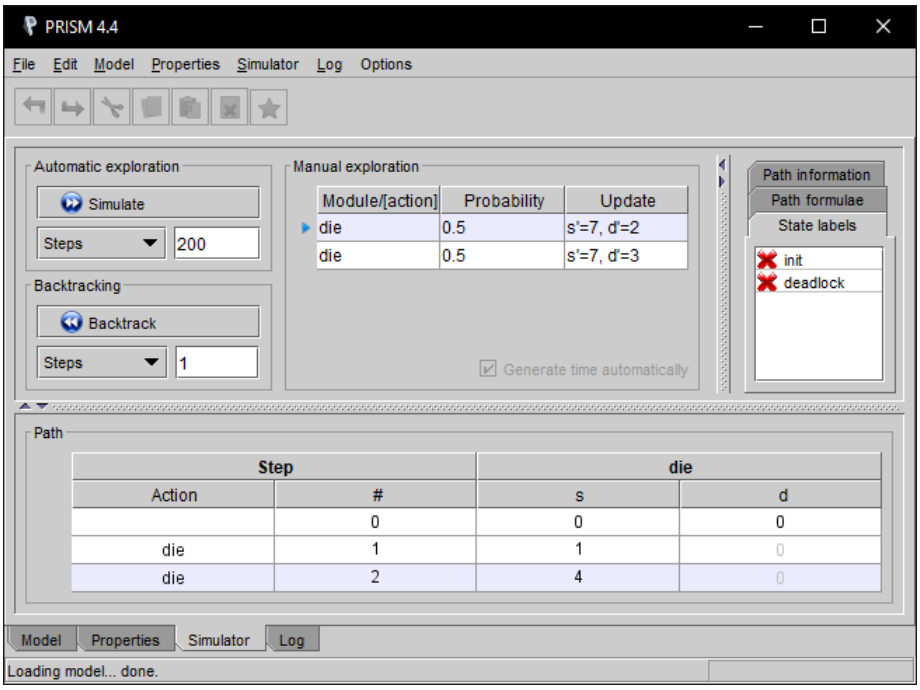

Abbildung 2.3: Die Simulationsumgebung von Prism

Properties

Prisms eigentliche Hauptaufgabe liegt in der Analyse und Evaluation von Stochastischen Modellen anhand von *properties*, also Eigenschaften, die das Modell haben soll. Properties werden in Prisms *property specification language* geschrieben. Diese fasst die viele probabilistische Logiken, wie PCTL und CSL zusammen. Mit diesen kurzen Statements kann man z.B Wahrscheinlichkeiten für das Erreichen eines bestimmten Zustands ermitteln oder herausfinden, wie viele *rewards* in einer bestimmten Anzahl Durchläufe gesammelt werden. Code 4 zeigt eine Property zur Analyse der Würfelemulation. Der Operator **P=?** zeigt, dass es sich um eine quantitative Wahrscheinlichkeitsanfrage handelt. Wir wollen also wissen, wie hoch die Wahrscheinlichkeit für unseren Pfad ist. Die Pfadeigenschaft wird dahinter in eckigen Klammern beschrieben. Der Operator **F** beschreibt einen Pfad, der **irgendwann** die gegebenen Bedingungen erfüllt. Diese sind in diesem Beispiel, dass s=7 und d=x sein müssen. Informell lautet diese Property also: **Wie groß ist die Wahrscheinlichkeit, irgendwann einen Zustand zu erreichen, in dem s=7 und d=x gilt.** Dass wir für x, also die gewünschte Augenzahl, eine Konstante einführen, erlaubt uns diese Property für alle Augenzahlen zu verwenden oder durch ein Experiment gleich mehrere abzudecken. Als Ergebnis erhalten wir dann einen Graph, der die Wahrscheinlichkeit in Abhängigkeit von x darstellt(Abbildung 2.4).Wir

```
1       const int x;
2       P=? [F s=7 & d=x]
```

Code 4: Prism Property zum Würfelbeispiel.

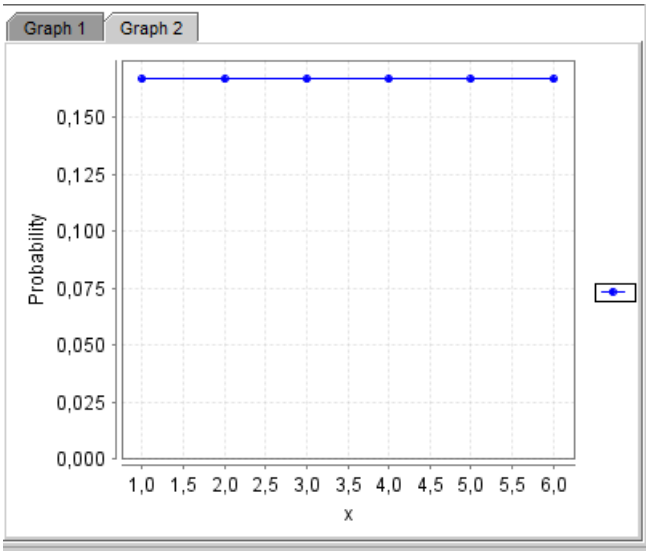

Abbildung 2.4: Ergebniss der Würfelemulation als Graph.

sehen, dass die Wahrscheinlichkeit der Augenzahlen erfolgreich emuliert wurde. Für jede Augenzahl x gilt $P(x) = \frac{1}{6}$.

Kapitel 3

Ziele dieser Arbeit

In diesem Kapitel schauen wir uns das, vom Fachgebiet Software Engineering konzipierte, Beispiel eines High Voltage Coupling Systems genauer an. An diesem Beispiel erläutern wir dann die Reliability Fragestellungen, die in dieser Arbeit diskutiert werden sollen.

3.1 Das HVCS

Das Modell des HVCS ist ein einfaches Beispiel aus dem ScenarioTools repository, welches zugleich im Tutorial benutzt wird [1]. Dabei wird spezifiziert, wie ein Benutzer eine Hochspannungsverbindung zwischen zwei Komponenten, etwa einem LKW-Zugfahrzeug und Anhänger, herstellt, ohne dass es zu gefährlichen Lichtbögen oder Ähnlichem kommt. Diese treten auf, wenn der Stecker eingesteckt oder entfernt wird, während Strom durch die Leitung fließt. Um dies zu verhindern, kann die Steckdose verriegelt werden. Ein Schema des benutzten HVCS sieht man in Abbildung 3.1.

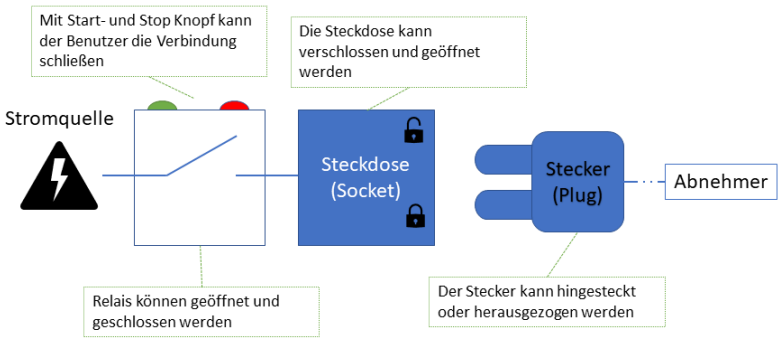

Abbildung 3.1: Schema eines HVCS

[1] http://scenariotools.org/tutorial-automotive-high-voltage-coupling-system/

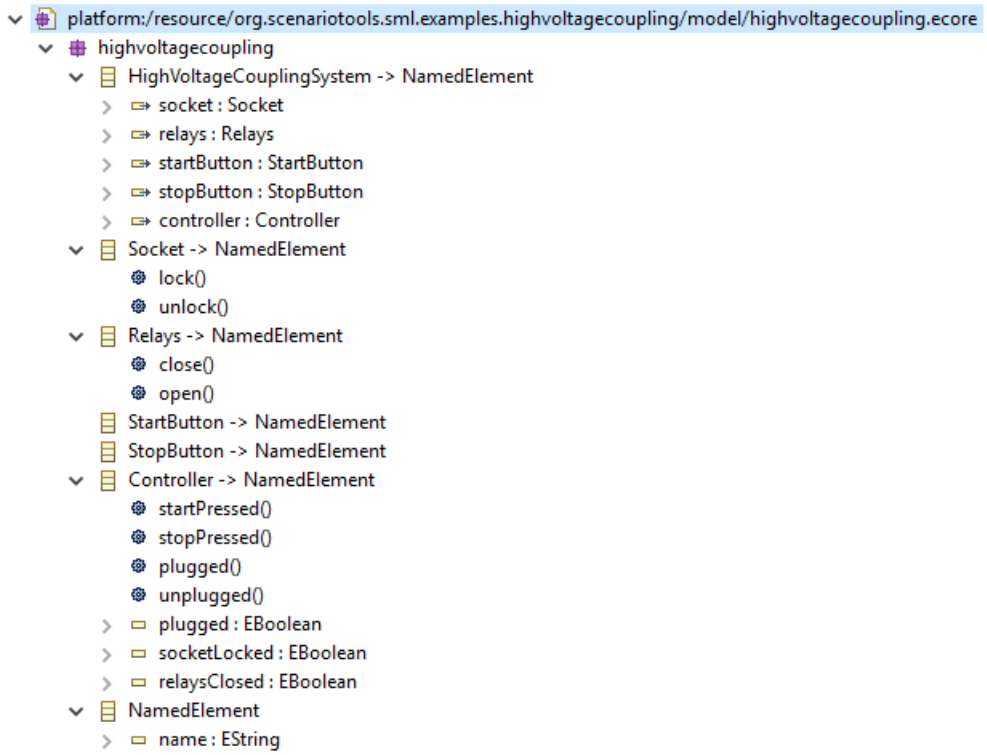

Abbildung 3.2: Das Klassendiagramm des HVCS

Aus diesem Schema erstellen wir nun einzelne Klassen mitsamt ihrer Attribute und Operationen. Daraus ergibt sich ein Ecore-Klassendiagramm, welches in Abbildung 3.2 zu sehen ist. Alle Klassen erben von einer Abstrakten Klasse *NamedElement*, damit sie ein Namensattribut erhalten. Die meisten Zuordnungen aus dem Schema sind trivial. Die neue Klasse *Controller* ist die Softwarekomponente, deren Verhalten wir spezifizieren und untersuchen wollen. Der Stecker wird modelliert, indem seine Operationen dem Controller zugesprochen werden. In einer SML-Spezifikation wird der Inhalt einer Nachricht stets als Operation des Empfängers modelliert. Daher kann der Controller nun Nachrichten mit dem Inhalt *plugged* oder *unplugged* erhalten. Eine eigene Klasse *Plug* wird nicht benötigt.

Aus diesem Klassendiagramm können wir nun, zusammen mit gesammelten Anforderungen an das System, eine SML-Spezifikation erstellen. Abbildung 5 zeigt einen Ausschnitt davon. Die komplette Spezifikation findet sich im Anhang 11. In dem Ausschnitt finden wir ausgesuchte Szenarien, die wir später als Beispiel für Reliability Fragen und unsere Lösungen zu diesen verwenden werden.

Wir können diese Szenarien in zwei semantische Teile unterteilen.

```
1
2     [Importieren des ecore Klassenmodells und setzen des Controllers als einzige
          kontrollierbare Klasse]
3
4     non-spontaneous events {
5     Controller.unplugged
6     }
7
8     collaboration PlugAndStart {
9
10    static role Socket socket
11    static role Relays relays
12    static role StartButton startButton
13    static role StopButton stopButton // added to allow unplugging
14    static role Controller controller
15
16
17    guarantee scenario WhenPluggedThenLock{
18        socket->controller.plugged()
19        strict urgent controller->socket.lock()
20    }
21
22    guarantee scenario SetPlugged{
23        socket->controller.plugged()
24        strict urgent controller->controller.setPlugged(true)
25    }
26
27    guarantee scenario SetLocked{
28        controller->socket.lock()
29        strict urgent controller->controller.setSocketLocked(true)
30    }
31
32
33    guarantee scenario WhenStartPressedThenCloseContact{
34        startButton->controller.startPressed()
35        interrupt [!controller.plugged || !controller.socketLocked || controller.
              relaysClosed]
36        strict urgent controller->relays.close()
37    }
38
39    guarantee scenario OnlyCloseRelaysWhenPlugPluggedAndLocked {
40        controller->relays.close()
41        violation [!controller.plugged || !controller.socketLocked]
42    }
43
44    guarantee scenario SetRelaysClosed{
45        controller->relays.close()
46        strict urgent controller->controller.setRelaysClosed(true)
47    }
48
49    /*
50    * Stop the flow of electricity to when pressing the
51    * stop button to allow unplugging
52    */
53    guarantee scenario OpenRelaysWhenStopping {
54        stopButton->controller.stopPressed()
55        interrupt [!controller.relaysClosed]
56        strict urgent controller->relays.open()
57    }
58
59    guarantee scenario ReleaseLockWhenStopping {
60        stopButton->controller.stopPressed()
61        interrupt [!controller.socketLocked]
62        strict urgent controller->socket.unlock()
63    }
64
65    guarantee scenario SetRelaysOpen {
66        controller->relays.open()
67        strict urgent controller->controller.setRelaysClosed(false)
68    }
69
70    guarantee scenario SetUnlocked {
71        controller->socket.unlock()
72        strict urgent controller->controller.setSocketLocked(false)
73    }
74
75
76    }
```

Code 5: Ausschnitt der HVCS Spezifikation

Bis einschließlich des Szenarios *SetRelaysClosed* wird das Kuppeln der Verbindung modelliert. Der Stecker wird hineingesteckt und die Steckdose verriegelt (Zeile 17-30). Danach kann durch Drücken des Startknopfes die Verbindung hergestellt werden (Zeile 33-47). Dabei ist zu beachten, dass unser Controller für den Verriegelungszustand der Steckdose und die Position des Steckers jeweils ein boolean Attribut besitzt. Mithilfe dieses wird in Zeile 35 geprüft, ob es sicher ist, die Relais zu schließen und somit Strom auf die Leitung zu geben. Konkret sorgt dort das *interrupt* Schlüsselwort für einen Abbruch des Szenarios, falls die Bedingung in den eckigen Klammern als wahr ausgewertet wird.
Der zweite Teil dieser Spezifikation stellt den umgekehrten Prozess dar. Also den Entkupplungsprozess. Dabei werden, nach drücken des Stopp Knopfes, die Relais geöffnet und die Steckdose entriegelt(Zeilen 53-62). Das Schlüsselwort *urgent* (Zeile 56 und 62) sorgt dafür, dass beides in einem sogenannten *Superstep* passiert. Dieser bezeichnet einen Zeitraum in dem nur das System agiert. Folglich müssen sowohl die Steckdose entriegelt, als auch die Relais entriegelt sein, bevor die Umwelt, also auch der Benutzer des HVCS, wieder agieren darf, indem sie z.B den Stecker herauszieht.
In Szenarien, die in dem Ausschnitt nicht zu sehen sind, werden weitere Assumptions beschrieben, wie die Annahme, dass der Benutzer den Stecker irgendwann herauszieht, oder dass dies nicht möglich ist, solange die Steckdose verriegelt ist; sowie Guarantees, die fordern, dass ein entfernen des Steckers bei geschlossenen Relais verboten ist.

3.1.1 Die Strategie

Unter Annahme unseres Klassendiagramms und der SMl-Spezifikation, sowie der passende Run Configuration kann ScenarioTools uns nun eine Abfolge von Systemaktionen, eine Strategie, erstellen. Diese sehen wir in Abbildung 3.3 oder, in etwas größer, im Anhang**??**. In der Darstellung sehen wir allerdings nicht jeden erreichten Zustand, sondern nur die umwelt-kontrollierten. Da das Verhalten des Systems in einer Strategie festeht, haben system-kontrollierte Zustände immer nur eine abgehende Transition. Auf diese Weise kann man die Strategie zusammenfassen und so lesbarer gestalten. An den, hier noch zu sehenden Transitionen steht nun jeweils eine Abfolge an verschickten Nachrichten. Diese startet immer mit einer Umwelt-Nachricht, da der Ausgangszustand Umwelt-kontrolliert ist.

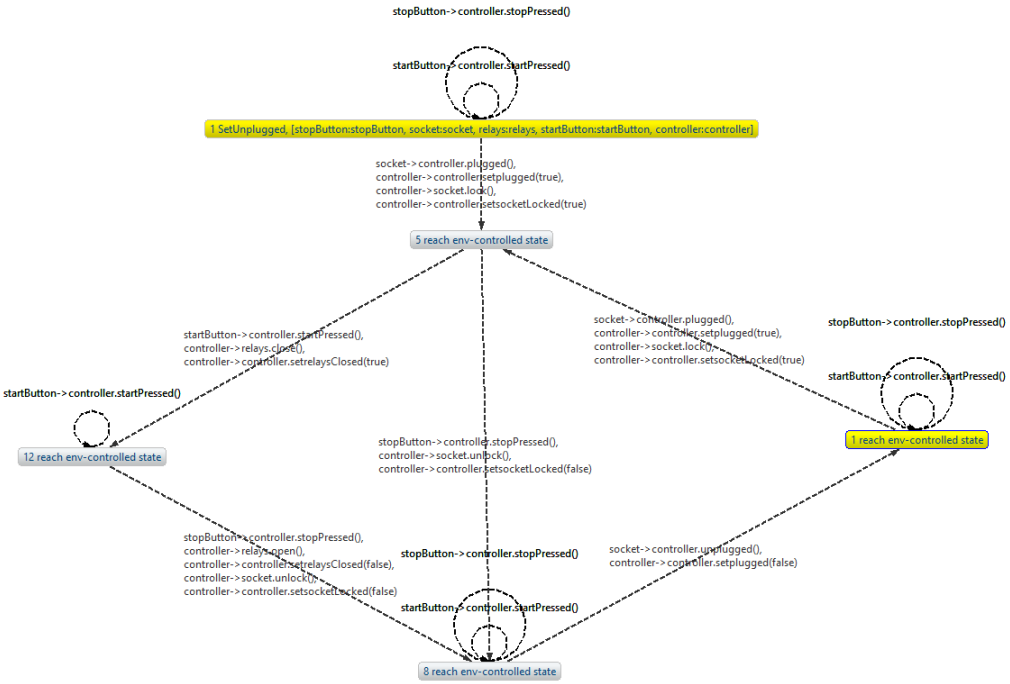

Abbildung 3.3: Aus der Spezifikation erzeugte Strategie für Systemaktionen im HVCS

3.2 Reliability Fragen

Wir wollen uns nun einige Fragestellungen aus dem Gebiet des Reliability Engineerings anschauen. Diese sind der Grund für die in dieser Arbeit erdachte Erweiterung von ScenarioTools . Eine gute Grundlage für eine Abschätzung der Zuverlässigkeit eines Systems ist die Betrachtung der **RAMS**. RAMS steht für **Reliability, Availability, Maintainability, Safety**. Diese Bereiche werden im Folgenden erklärt, eingeordnet und Fragestellungen am Beispiel erläutert.

3.2.1 Reliability

Reliability beschreibt die Fähigkeit eines Systems, seine korrekte Funktion über einen gewissen Zeitraum unter bestimmten Bedingungen aufrecht zu erhalten. Eine Metrik zur Reliability Abschätzung ist die *Mean-Time-To-Failure* (MTTF), also die mittlere Zeit bis zum Ausfall des Systems. Mit ihr kann man benötigte Wartungsintervalle festlegen und gegebenenfalls die Qualität der verwendeten Komponenten anpassen, sollte die MTTF des Systems von der gewünschten abweichen. Unser HVCS Beispiel beinhaltet mehrere Komponenten, die vor allem nach gewisser Zeit, Fehler produzieren können. Ein Relais könnte irgendwann nicht mehr korrekt schalten, die

Steckdose nicht mehr korrekt verriegeln. Wir wollen gerne schon in der Anforderungserhebung mit ScenarioTools wissen, wann das für die jeweiligen Komponenten oder das Gesamtsystem eintritt.

3.2.2 Availability

Unter Availability verstehen wir die Verfügbarkeit des Systems. Diese hängt sowohl von der Reliability des Systems, als auch von seiner Maintainability ab. Wenn wir die Availability unseres HVCS genau abschätzen können, können wir Entscheidungen über eventuell nötige Redundanzen zu unserem System fundiert treffen. Das ermöglicht Kosteneinsparungen. Um dies zu realisieren, benötigen wir sowohl Kenntnisse über die nötigen Wartungsintervalle, als auch über die jeweils nötige Zeit pro Wartung.

3.2.3 Maintainability

Die Maintainability eines Systems hängt von mehreren Faktoren ab: Wie leicht man einzelne Komponenten erreichen uns auswechseln kann; Wie modular das System aufgebaut ist; sind die Teile standardisiert oder Einzelanfertigungen. Ist sie uns bekannt, können wir die *Downtime*, also die Zeit in der das System funktionsunfähig ist, abschätzen und Wartungskosten errechnen.

3.2.4 Safety

Für ein, mit Menschen interagierendes, System, wie dem HVCS, ist die Safety, also die Sicherheit, eine überaus wichtige Eigenschaft. Wenn der Stecker im Beispiel durch einen aufgetretenen Fehler doch einmal gezogen werden kann, obwohl die Verbindung gerade Strom führt, kann es zu lebensgefährlichen Lichtbögen kommen. Auch in anderen Bereichen, in denen es z.B um die Arbeit mit automatisierten Fertigungsanlagen geht, ist Sicherheit wichtig. Wie oft kommt es durch einen Ausfall eines Komponenten zu gefährlichen Situationen? Was passiert, wenn ein Fehler auftritt? Beschränkt er sich auf eine Komponente oder löst er weitere Fehler, über das System verteilt aus? Wenn wir Fehler intuitiv in SML modellieren und mit ScenarioTools und Prism auswerten können, können wir sicherere Gesamtsysteme schaffen und Unfälle reduzieren.

3.2.5 Einordnung in diese Arbeit

Wir beschäftigen uns in dieser Arbeit nicht mit allen vier RAMS Eigenschaften. Unser Hauptziel ist, mit Hilfe des PA Rewardsystems Safety-Eigenschaften zu modellieren, sowie Design-Reliability zu untersuchen. Außerdem wollen wir untersuchen, ob eine Abschätzung der MTTF mit unserem Ansatz möglich ist.

Kapitel 4

Ausarbeitung der ScenarioTools Erweiterung

In diesem Kapitel schauen wir uns die Erweiterungen an ScenarioTools und der Scenario Modelling Language an, um die Fragestellungen des vorigen Kapitels beantworten zu können. Zuerst werden wir dabei auf die nötigen Syntaxänderungen in SML eingehen, um dann über allgemeine Änderungen an ScenarioTools zur Konvertierung der Spezifikation nach Prism überzugehen. Wir werden jeden Aspekt dieses Prozesses sowohl von einem Design Standpunkt, als auch anhand des, in 3.1, vorgestellten Beispiels beleuchten.

4.1 Erweiterungen in SML

Um Aussagen über das Verhalten des Gesamtsystems in Bezug auf probabilistische Ereignisse beziehungsweise Nachrichten treffen zu können, müssen wir zunächst SML um eine Möglichkeit erweitern, einzelne Nachrichten mit Wahrscheinlichkeiten zu versehen, ohne dabei die normale Verwendung von SML einzuschränken.Unser Ansatz dafür sieht folgendermaßen aus: Ist eine Nachricht in SML von einer Wahrscheinlichkeit abhängig, so gibt es immer mindestens zwei Möglichkeiten in Form zweier verschiedener Nachrichten, die als Ergebnis, bzw. als nächste Nachricht eintreten können. Dieses wird anhand eines normalen sechs-seitigen Würfels schnell deutlich. Werfen wir den Würfel, können verschiedene Augenzahlen, jeweils mit einer bestimmten Wahrscheinlichkeit, auftreten.
Für dieses Verhalten, also die Auswahl eines Ergebnisspfades, gibt es in SML schon eine Kontrollstruktur - Die *Alternative*. Sie ermöglicht bereits bedingte Verzweigungen. Parallel dazu führen wir nun eine neue Kontrollstruktur ein. Die *probabilistic Alternative*. Auch sie beschreibt eine Verzweigung, allerdings eine wahrscheinlichkeitsbasierte. In Code 6 sehen wir eine mögliche Modellierung des Würfelbeispiels in SML. Dabei handelt es sich nicht um

```
1
2  guarantee scenario RollTheDice{
3  //Die \textit{Initiator-Nachricht}
4  controller -> die.roll()
5  probabilistic alternative
6      //case 1
7      [1/6;0]{
8      die->controller.rolled(1)//resolution
9      }or
10     //case 2
11     [1/6;0]{
12     die->controller.rolled(2)
13     }or
14     //case 3
15     [1/6;0]{
16     die->controller.rolled(3)
17     }or
18     //case 4
19     [1/6;0]{
20     die->controller.rolled(4)
21     }or
22     //case 5
23     [1/6;0]{
24     die->controller.rolled(5)
25     }or
26     //case 6
27     [1/6;0]{
28     die->controller.rolled(6)
29     }
30 }
```

Code 6: Ein Würfelwurf als probabilistic alternative

eine vollständige SML-Spezifikation, sondern nur um ein Szenario, welches die Idee hinter der probabilistic alternative(PA) näher bringen soll.

Wir sehen darin die Funktionsweise der PA. Eine Initiator-Nachricht stellt eine Anfrage an die Umwelt, in diesem Fall den Würfel, dar. Mithilfe der PA kann auf diese eine Antwort modelliert werden. Dazu benutzen wir, wie in der normalen alternative, *cases*, also Fälle, in denen Interaktionen, also wiederum Blöcke von Nachrichten und Kontrollstrukturen, stehen können. Im Würfelbeispiel haben wir 6 Fälle: Je einen für jede Seite des Würfels. Getrennt werden die cases durch das *or* Schlüsselwort. Vor der Interaktion innerhalb eines cases steht in eckigen Klammern jeweils die Wahrscheinlichkeit für diesen Fall, hier immer $\frac{1}{6}$, sowie eine arbiträre *reward*, die später bei der Analyse des Modells verwendet wird, um Aussagen über den Nutzen dieses Falles zu treffen. Die Wahrscheinlichkeit kann als beliebiger double angegeben werden. Die tatsächliche Wahrscheinlichkeit wird dann durch Anteil der case-Wahrscheinlichkeit an der Summe der Wahrscheinlichkeiten bestimmt. Sei P die gesuchte Wahrscheinlichkeit für den Case i, p_i die in SML eingetragene Wahrscheinlichkeit und N die Anzahl der Cases, dann gilt:

$$P(i) = \frac{p_i}{\sum_{n=1}^{N} p_n}$$

Die Initiator-Nachricht ist optional und beeinflusst die Auswertung der PA

nicht, dient allerdings der Lesbarkeit.Ein Szenario kann auch nur aus einer PA bestehen. Die semantischen Auswirkungen beschreiben wir später in diesem Kapitel. Die erste Nachricht jedes cases nennen wir *resolution*. Sie gibt eine Antwort auf die im Initiator gestellte Frage.
Um dieses Verhalten in SML zu ermöglichen müssen auch formale Änderungen an der Sprache durchgeführt werden. Code 7 zeigt eine PA in XText als Bestandteil einer Interaction innerhalb einer Collaboration. Wir beschreiben die PA dabei zunächst als eigenes *InteractionFragment* anstatt die Alternative zu erweitern. Die erweiterte Collaboration findet sich im Anhang[12].

```
1   ProbabilisticAlternative returns ProbabilisticAlternative:
2       {probabilisticAlternative} 'probabilistic' 'alternative'
3       cases+=ProbabilisticCases ('or' cases+=ProbabilisticCases)+;
4
5   ProbabilisticCase returns ProbaibilisticCase:
6       {case}
7       '['probability = DOUBLE ';' reward = DOUBLE ']'
8       caseInteraction=Interaction;
```

Code 7: Die PA als XText-Definition

4.1.1 Probabilistic Alternatives im HVCS

Nachdem wir uns nun die Idee und Theorie der PA angeschaut haben, wollen wir nun das HVCS erweitern, um die ,in Kapitel 3.2, vorgestellten Ziele zu verwirklichen. Wir fügen dazu PAs in den Teil der Spezifikation ein, der sich mit dem Herstellen der Verbindung befasst. Wir gehen außerdem davon aus, dass neu eingeführte Methoden auch im Klassendiagramm ergänzt wurden.
Wir realisieren zuerst eine Fehlermodellierung der Steckdose. Diese soll mit einer Wahrscheinlichkeit von 1% **nicht** verschließen. Nehmen wir an, der Controller wüsste, wann die Steckdose verschlossen wurde und wann nicht. Dann können wir ein Fehlschlagen des Verschlussvorgangs durch Änderungen im *SetLocked* Szenario wie folgt vornehmen:

```
1   guarantee scenario SetLocked{
2       controller->socket.lock()
3       probabilistic alternative
4       [0.99;1]{
5           strict committed socket->controller.setSocketLocked(true)
6       }or
7       [0.01;-10]{
8           strict committed socket->controller.setSocketLocked(false)
9           committed controller->socket.lock()
10      }
11  }
```

Wir verwenden hier und in den weiteren Beispielen den Reward-Parameter zur Modellierung von Nachteilen. Eine höhere Reward stellt hier einen Nachteil für den Systemablauf dar. Die Namenswahl ist also durchaus suboptimal. Begründet wird sie dadurch, dass es schon eine Struktur in SML gibt, die *Cost* benutzt und von der wir uns, zumindest zunächst, abgrenzen wollen. Der Begriff Rewards stammt aus Prism und wird dort synonym mit

Cost verwendet. Wie mit Rewards modelliert wird, bleibt letztendlich dem Benutzer von ScenarioTools überlassen. Wir sehen an diesem Beispiel auch, dass die Verwendung von Safety- und Liveness-Schlüsselwörtern auch innerhalb einer PA möglich ist. Wir erinnern uns außerdem, dass die Resolution-Nachrichten immer Umwelt-Nachrichten sein müssen, weshalb die *setSocketLocked()* Nachrichten (Zeile 5&8)nun vom Socket verschickt werden. Da die Resolutionen normale Nachrichten sind, können sie auch als solche verwendet werden. Ein neues Szenario könnte z.B durch die Nachricht in Zeile 8 aktiviert werden, um mit dem Fehler umzugehen. Genauso könnte das auch direkt in dem Case weiter behandelt werden. Im Beispiel versuchen wir einfach erneut Um das zu verdeutlichen schauen wir uns eine Erweiterung des *SetRelaysClosed* Szenarios an. Es sei in 10% der Fälle nicht möglich, die Relais zu schließen und so eine Stromführende Verbindung herzustellen.

```
1     guarantee scenario SetRelaysClosed{
2         controller->relays.close()
3         probabilistic alternative
4         [90;0]{
5         committed relays->controller.setRelaysClosed(true)
6         }or
7         [10;5]{
8         committed relays->controller.setRelaysClosed(false)
9
10        }
11  }
```

Das Beispiel demonstriert noch einmal, dass die Wahrscheinlichkeiten nicht zwangsläufig als Dezimalzahlen mit Summe Eins angegeben werden müssen. Die Reward für den Fehlerfall ist hier nur -5, da ein nicht geschlossenes Relais ungefährlicher ist, als eine unverschlossene Steckdose.

4.2 Anpassen der ScenarioTools Simulationsumgebung

Damit wir unsere erweiterte Spezifikation mit dem ScenarioTools *Play-out* Algorithmus simulieren können müssen wir kleine Änderungen an der Simulationsumgebung vornehmen. Da wir die Auswertung und alle dafür notwendigen stochastischen Berechnung nach Prism ausgelagert haben, sind diese aber eher kosmetischer Natur. Da die *alternative* bereits existiert, können wir die Behandlung der PA im Play-out davon übernehmen. Sobald wir die PA im Ablauf erreichen, werden alle Resolution-Nachrichten aktiv und können ausgewählt werden. Entscheidet sich der Benutzer für eine, zu einer PA gehörenden, Nachricht, wird mit der Interaktion des gewählten Cases fortgefahren.
Die *Event Selection View*, in der bei der Simulation die erwarten Nachrichten angezeigt und die nächste Nachricht ausgewählt werden kann, soll im Fall einer PA auch die entsprechenden Wahrscheinlichkeiten anzeigen. Dazu fügen wir hinter den betroffenen Nachrichten eine Spalte *Probability*

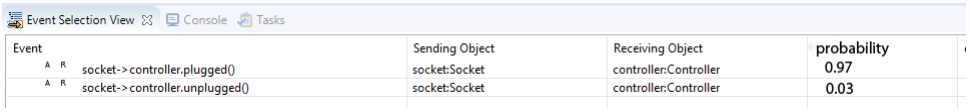

Abbildung 4.1: Prototyp der Eventselection mit Wahrscheinlichkeitsangaben für PA Resolution-Nachrichten.

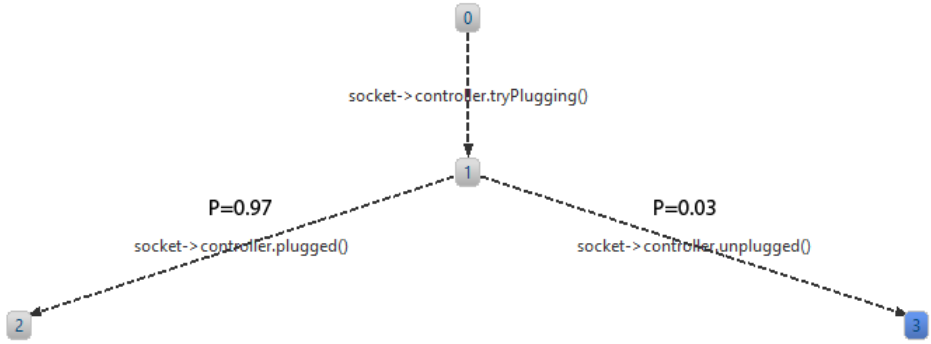

Abbildung 4.2: Prototyp des erweiterten Simulationsgraphen mit annotierten Wahrscheinlichkeiten

ein(Abbildung 4.1). Dort wird bereits die berechnete Wahrscheinlichkeit angezeigt.

Außerdem fügen wir den entsprechenden Transitionen im Simulationsgraphen Wahrscheinlichkeiten an. So kann man, nachdem man einen gewünschten Pfad simuliert hat, nachverfolgen, wie wahrscheinlich die manuell gewählten Nachrichten waren(Abbildung 4.2).

Ob eine PA direkt aufgelöst wird, oder ob wir die Möglichkeit haben zu warten oder andere Nachrichten zu verschicken, hängt davon ab ob die Resolutionen mit dem Schlüsselwort *committed* versehen sind. In Abbildung 4.3 sehen wir, dass eine *startPressed* Nachricht versandt wird, bevor die PA aufgelöst wird. Dieses Verhalten erlaubt das Modellieren von zeitaufwändigen Entscheidungsprozessen, wie dem Herstellen einer Netzwerkverbindung. Bis bekannt ist, ob die Verbindung erfolgreich hergestellt wurde, können so noch andere Events eintreten . Abbildung 4.3 zeigt eine solche Konfiguration, in der sowohl die PA, als auch ein nicht-probabilistisches Event eintreten können. ScenarioTools ignoriert an dieser Stelle alle Wahrscheinlichkeiten und behandelt die PA als normale Alternative, also als zwei Umweltevents, sodass wir insgesamt drei mögliche Umweltevents haben. Prism hingegen berechnet, wie wir später sehen werden, die Wahrscheinlichkeiten neu, sodass die PA Cases im Verhältnis zueinander ihre Wahrscheinlichkeiten behalten, die PA insgesamt aber nur mit einer Wahrscheinlichkeit von 50% aufgelöst

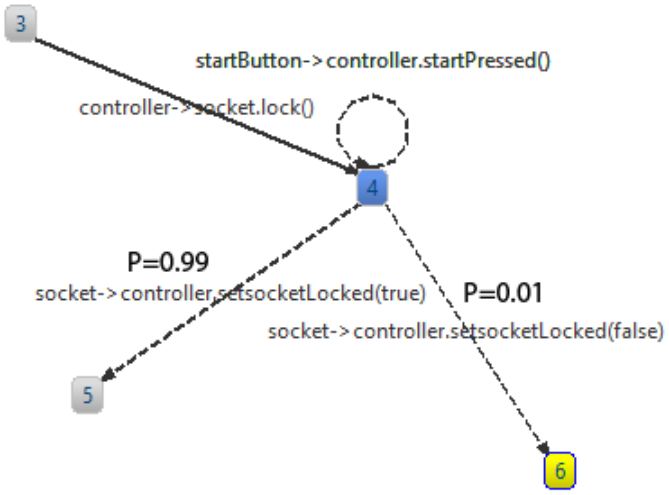

Abbildung 4.3: Der Simulationsgraph des *setLocked* Szenarios zeigt die Möglichkeit zwischen Initiator und Resolution andere Nachrichten zu verarbeiten.

wird (Bei einem anderen Event). In der in Abbildung 4.3 zu sehenden Event-Konfiguration wären die Wahrscheinlichkeiten also wie folgt verteilt:

- P(startButton->controller.startPressed()) = 0.5
- P(socket->controller.setSocketLocked(true)) = $0.5 * 0.99 = 0.495$
- P(socket->controller.setSockeLocked(false)) = $0.5 * 0.01 = 0.005$

Die Indifferenz von ScenarioTools bezüglich des Probabilismus in solchen Event-Konfigurationen erklärt auch, warum es möglich ist, Szenarios nur aus einer PA zu modellieren. Dann hat die Umwelt zu jedem Zeitpunkt, an dem sie normalerweise auch die Möglichkeit hätte, eine Nachricht zu verschicken, eben auch die Möglichkeit die, am Anfang eines Szenarios stehende, PA aufzulösen.

4.3 Controller Synthese mit PA

Wenn wir aus einer mit PAs erweiterten Spezifikation einen Controller synthetisieren wollen, funktioniert dies, solange die Wahrscheinlichkeiten ignoriert werden, genauso wie vorher. Da alle Resolutionen von Umwelt-Nachrichten sind, das System sich also die Auflösung einer PA nicht aussuchen darf, muss es in der Synthese auf alle möglichen Cases reagieren. Genau wie eine in einer normalen *alternative*, in der die Initialisierungsnachricht von der Umwelt stammt.

Es existiert bereits ein Algorithmus zur Kosten-optimierten Controller-Synthese[4]. Dieser hat zwar noch nichts mit probabilistischen Szenarien zu tun, es ist aber denkbar in der Zukunft eine ähnliche Optimierung auch für Rewards und Wahrscheinlichkeiten zu implementieren, bzw. den vorhandenen Algorithmus um Wahrscheinlichkeiten zu erweitern.

4.4 Prism Modelle generieren

Das Ziel dieser Arbeit ist es, SML-Spezifikationen mit Hilfe von probabilistischen Simulationen auf Reliability Fragestellungen zu untersuchen. Dies verwirklichen wir, indem wir synthetisierte Strategien aus ScenarioTools nach Prism exportieren. Aus einer Strategie wird dabei ein DTMC-Modell in Prism generiert, welches die gleichen Zustände und Transitionen wie die Strategie hat. Außer den Transitionen benötigen wir noch die Anzahl der Zustände, welche ebenfalls aus der Strategie zu beziehen sind. Mit diesen Informationen können wir nun ein Prism Modell erstellen. Hat die Strategie z.B. eine Transition von Zustand **1** nach **2** mit der Nachricht *sender->receiver.Op1()*, wandeln wir diese wie folgt in ein Prism *command* um:

```
1    //s wird als Variable fuer den aktuellen Zustand benutzt.
2    [sender_TO_receiver_Op1] s=1 -> (s'=2);
```

Das *Action* Feld ([]), welches wir als Kommentarfeld benutzen, erlaubt nur sehr wenige Sonderzeichen, sodass wir den SML-Nachrichten-Pfeil durch ein *_TO_* emulieren. Ist der Ausgangszustand ein Initiator einer PA, wie in Abbildung 4.4, erweitern wir das *Update* des Prism Commands, also den Teil rechts des Pfeils, um die anderen Transitionen und versehen alles mit den gegebenen Wahrscheinlichkeiten:

```
1    [receiver_opA1__receiver_opA2] s=0 -> 0.6 : (s'=1) + 0.4 : (s'=2);
2    [continue]
```

Um später die Rewards den einzelnen Zuständen so zuordnen zu können, dass sie nur eingesammelt werden, wenn man über die entsprechende PA in den Zustand gelangt, fügen wir für jeden Case einen eigenen Zustand ein. Da wir die Zustandsmenge und damit die Anzahl der Zustände der Strategie kennen, können wir Kollisionen der Nummern vermeiden. Ein weiteres dabei auftretendes Problem ist die Unfähigkeit Prisms, mehrere Commands mit gleichem Guard zu einem zusammenzufassen. Es ist uns also nicht möglich für jede Transition ein eigenes Command und damit auch einen eigenen Kommentar zu verfassen, da dann die Wahrscheinlichkeiten innerhalb der commands nicht die Summe eins haben. Um die Transitionen trotz dessen identifizieren zu können, schreiben wir die Empfängerseiten der PA-Resolutionen, durch *__* getrennt, in den Kommentar. Die Reihenfolge entspricht der, in der sie auch im Prism Simulationsfenster angezeigt werden. Dieses Vorgehen ist durch die Größe des Simulationsfensters beschränkt. Ab einer bestimmten Länge ist der Kommentar dort nichtmehr vollständig

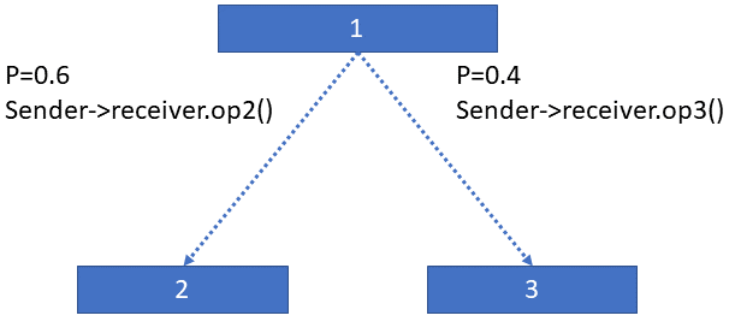

Abbildung 4.4: Schematische Strategie einer PA; Die Zahlen in den Kästen sind die Zustandsnummern.

lesbar. Diese hängt natürlich auch von der Länge der SML-Operationen und Empfängernamen ab. Zur Not korrespondieren die Zustandsnummern immer mit denen der Strategie, sodass man auch dort nachverfolgen kann, welche Transition man gerade simuliert. Sobald die SML-Spezifikation und damit eventuell auch ihre Strategie eine gewisse Komplexität überschreitet, sollte man sowieso nicht mehr von Hand in Prism simulieren, sondern komplett auf Properties vertrauen, welche wir in Kapitel 5 behandeln.
Gehen von einem Zustand sowohl PAs als auch normale Transitionen ab, konvertieren wir beide unabhängig voneinander, wie bisher beschrieben. Prism errechnet dann die tatsächliche Wahrscheinlichkeit für jede Transition. Am Ende des Prism Modells fügen wir dann für jeden PA Case die entsprechenden Rewards in eine Rewardstruktur ein, in dem wir den, für jeden Case erzeugten, Extrazustand mit der in SML-spezifizierten Reward annotieren. Als Beispiel haben wir die um die Zwei vorgestellten PAs erweiterte Spezifikation des HVCS in Prism von Hand nach Prism konvertiert(Code **??**). Es wurden nur die Umwelt-kontrollierten Zustände konvertiert, da sie die PAs enthalten. Ein automatisierter Konvertierer würde allerdings alle Zustände und Transitionen aus der Strategie in ein Prism Modell umwandeln, damit keine Informationen verloren gehen.

```
1
2  dtmc
3
4
5  module hvcs
6  // state Strategy is in
7  s : [0..25] init 1;
8  i : [0..1000] init 0;
9
10 [startbutton_TO_controller_startPressed] s=1 -> (s'=1);
11 [stopButton_TO_controller_stopPressed] s=1 -> (s'=1);
12 [socket_TO_controller_plugged] s=1 -> (s'=4);
13
14 //PA ob das Socket verschlossen wird
15 [controller_setSocketLocked_true__controller_setSocketLocked_false] s=4 -> 0.99 :
       (s'=15) + 0.01 : (s'=16);
16 [continue] s=15 -> (s'=6);
17 [continue] s=16&i<1000 -> 1 :(s'=4)&(i'=i+1);
18
19 [startbutton_TO_controller_startPressed] s=4 -> (s'=4);
20 [stopButton_TO_controller_stopPressed] s=4 -> (s'=4);
21 [stopButton_TO_controller_stopPressed] s=6 -> (s'=9);
22 [startButton_TO_controller_startPressed] s=6 -> (s'=12);
23 [startbutton_TO_controller_startPressed] s=9 -> (s'=9);
24 [stopButton_TO_controller_stopPressed] s=9 -> (s'=9);
25
26 //eingeschobener Endzustand
27 [socket_TO_controller_unplugged] s=9 -> (s'=0);
28 [continue] s=0 -> (s'=1);
29
30 //PA ob die Relais verschlossen werden
31 //s=15 ist ein eingeschobener Zustand, um die Rewardmodellierung zu ermoeglichen
32 [controller_setRelaysClosed_true__controller_setRelaysClosed_false] s=12 -> 0.9 :
       (s'=17) + 0.1 : (s'=18) ;
33 [continue] s=17 -> (s'=13);
34 [continue] s=18&i<1000 -> (s'=6) &(i'=i+1);
35
36 [stopButton_TO_controller_stopPressed] s=13 -> (s'=9);
37
38
39
40 endmodule
41 label "Ziel" = s=0;
42 label "error" = s=16 | s=18;
43 rewards
44 s=15: 0;//Socket erfolgreich verschlossen
45 s=16 : 10;//fehlgeschlagen
46 s=17 : 0;//Relais erfolgreich geschlossen
47 s=18 : 5;//fehlgeschlagen
48 endrewards
49
50 rewards "numSuccesses"
51 s=0 : 1;
52 endrewards
```

Code 8: Das probabilistische HVCS Beispiel als Prism Modell; Die zwei eingefügten PAs sind mit Kommentaren gekennzeichnet.

Kapitel 5

Modellanalyse mit Prism Properties

Prism bietet uns die Möglichkeit unser generiertes Modell mit Hilfe von *Properties* zu analysieren. Diese ,in der *Property Specification language* erstellten, Eigenschaften liefern nach ihrer Auswertung Einblicke in die RAMS Eigenschaften unseres Systems. In diesem Kapitel wollen wir mögliche Properties betrachten und am HVCS-Beispiel zeigen, sowie einen Code-Generator diskutieren, der das Erstellen von Properties aus SML-Spezifikationen ermöglicht.

5.1 Beispielanwendungen

Wir schauen uns nun zunächst einige Beispiel-Properties zur Analyse des um PAs erweiterten HVCS Beispiels, beziehungsweise des zugehörigen Prism Modells an. Dabei versuchen wir die RAMS Fragestellungen aus Kapitel 3.2 zu beantworten. Das erweiterte HVCS Modell in Prism sehen wir in Code 8. Dabei ist zu beachten, dass wir nur Umwelt-kontrollierte Zustände betrachten und nach Prism konvertieren. Außerdem behandeln wir bereits alle Fehlerzustände in der Spezifikation, sodass es nie zu einem Abstürzen des Systems kommt. Wir behandeln die beiden unerwünschten PA Cases trotzdem als Fehler.

5.1.1 Reliability

Die Fehlerwahrscheinlichkeiten der Einzelkomponenten müssen uns bereits bekannt sein, um die Spezifikation erstellen zu können. Nun können mir mit Hilfe der Properties die Reliability des Gesamtsystems analysieren. Wir können dazu einen oder mehrere Zielzustände definieren. Wir können label auch benutzen, um Fehlerzustände anzuzeigen. In unserem Beispiel würden wir z.B. das Erreichen des Zustands $s = 0$ als Ziel modellieren. Dann haben

wir den kompletten Systemablauf einmal durchlaufen und stehen vor dem Beginn des nächsten Durchgangs. Mit der Property

```
1    P=? [F "ziel"]
```

können wir die Wahrscheinlichkeit P für das Erreichen unseres Zielzustandes berechnen. **P=?** besagt, dass wir eine Wahrscheinlichkeit suchen. In den eckigen Klammern steht immer die *Path Property*, oder Pfadeigenschaft, also ein auszuwertender Ausdruck. Hier verwenden wir den Typ **F**, der besagt, dass wir *eventually* die dahinterstehende Eigenschaft erfüllen. In unserem Fall ist diese das label SZieläls wahr auszuwerten. Diese Property wird zu 1 evaluiert, was nicht weiter überrascht, da wir bereits in der Strategie sehen, dass der Graph stark zusammenhängend ist, wir also jeden Zustand von jedem andern erreichen können. Sie prüft, ob für alle Zustände gilt, dass die Wahrscheinlichkeit den Zielzustand zu erreichen gleich oder größer Eins ist. Wir erreichen also zunächst einmal unseren Zielzustand. Die Frage ist, ob wir auf dem Weg dorthin keinen Fehlerzustand passieren. Wir definieren zunächst unsere Fehlerzustände ($s = 16 \& s = 18$) und evaluieren dann die Property:

```
1    P=? [!"error" U "Ziel"]
```

Das Ergebnis (0.9405) beschreibt die Wahrscheinlichkeit, das Ziel zu erreichen, ohne einen der genannten Fehlerzustände anzunehmen. Dabei handelt es sich um die Design-Reliability, also die Zuverlässigkeit des Systems im Neuzustand. Um den Pfad zu beschreiben nutzen wir hier den **U** Operator. Mit ihm können wir auswerten, ob der Operand links von ihm wahr ist, bis der Operand rechts wahr wird.
Wollen wir die Mean-Time-To-Failure analysieren, stoßen wir auf ein Problem: Sowohl ScenarioTools als auch unser DTMC-Modell sind zeitdiskret, sodass wir keine konkrete Zeitangabe treffen können. Außerdem modelliert unser Prism Modell keinen Verfall der Komponenten, sondern geht davon aus, dass sie immer mit der gleichen Wahrscheinlichkeit fehlschlagen. Wir können allerdings eine Variable 'numErrors' einführen, die für jeden Fehlerzustand, den wir annehmen erhöht wird. Nun vergeben wir eine Reward für das Erreichen des Zielzustandes. Wir können nun die Durchläufe zählen, die bis zu einer bestimmten Anzahl Fehler auftreten.

```
1    R{numSuccesses}=? [F numErrors >= x]
```

Code 9: Property für die erwartete Anzahl Durchläufe, bis x Fehler aufgetreten sind.

In einem Experiment können wir unsere Fehlertoleranz variieren und erhalten den in Abbildung 5.1.1 zu sehenden Graphen.

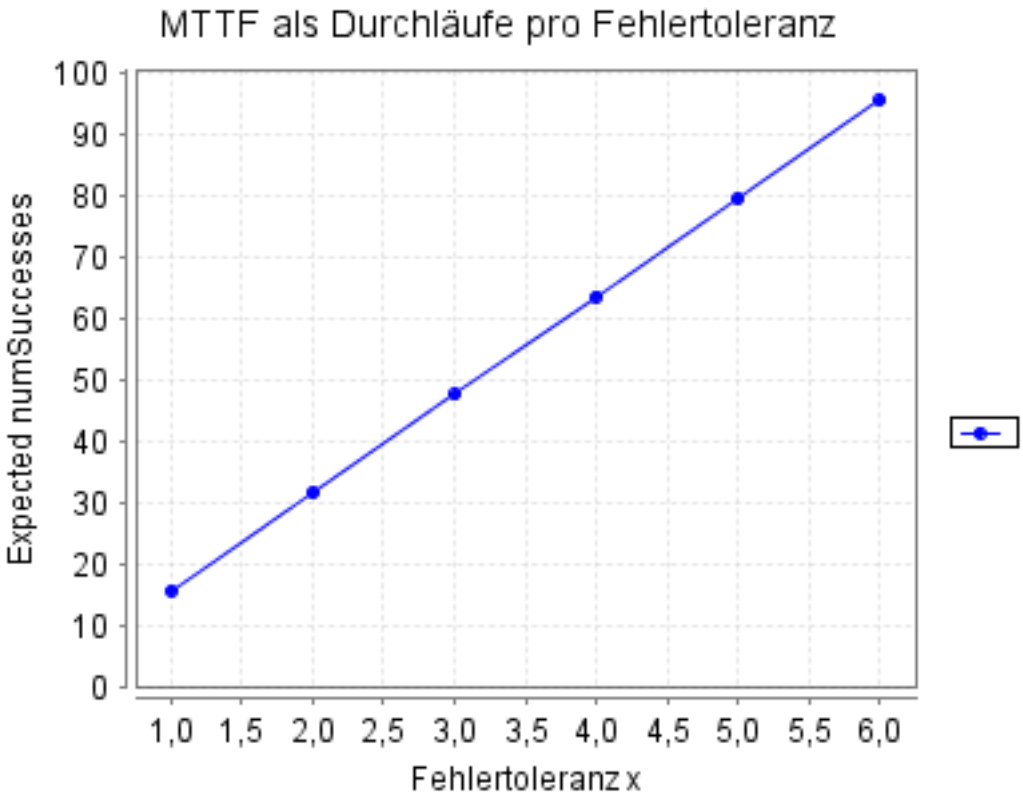

Abbildung 5.1: Ausgewertete MTTF abhängig von der tolerierten Fehleranzahl

5.1.2 Safety

In unserem Beispiel haben wir den Rewards Parameter der PA stets so modelliert, dass eine niedrigere Reward für mehr Sicherheit steht. Nun wollen wir die Gesamtsicherheit unseres Systems über die Rewards analysieren. Dabei müssen wir stets beachten, dass die vergebenen Rewards selbstgewählt sind und damit von der jeweiligen Modellierung abhängen. Folgende Property liefert uns die erwarteten gesammelten Rewards bis zum Zielzustand.

```
1    R=? [F "Ziel"]
```

Im Beispiel erreichen wir ungefähr 0.36 Rewards pro Durchlauf. Anhand dieser Zahl können wir nun die Sicherheit unseres Systems beurteilen indem wir sie mit anderen Spezifikationen für unser System vergleichen. Dabei müssen wir darauf achten, dass die Rewards der Spezifikationen immer gleich verwendet werden und auch die Gewichtung der Fehlerzustände, also die Größe der Rewards gleich ist. Natürlich können wir die, mit Prism ausgewertete, Sicherheit unseres Systems verbessern, indem wir die Rewards für Fehlerzustände verringern. Das sollte in der Praxis möglichst vermieden werden.

5.2 Properties aus ScenarioTools

Um dem Nutzer die aufwendige Einarbeitung in Prism und die property specification language zu ersparen, suchen wir nach Wegen Properties aus SML-Code oder der Benutzeroberfläche von ScenarioTools zu generieren.

5.2.1 Properties in SML

Eine Möglichkeit Properties zu spezifizieren ist, diese direkt in die SML-Spezifikation zu schreiben. Dazu können wir, ähnlich wie bei der Kostenoptimierung. ein Szenario erstellen, welches die zu analysierende Zustandsmenge durch seine Aktivität in jedem dieser Zustände darstellt.

```
1    guarantee scenario ProbabilisticEvaluation
2    analyze MTTF{
3        socket->controller.plugged()
4        socket->controller.unplugged()
5    }
```

Code 10: Beispiel einer MTTF Property in SML

Code 10 zeigt die in Code 9 vorgestellte Property als SML-Szenario. Wir führen das Schlüsselwort *analyze* als Parallele zu *optimize* ein. Dahinter schreiben wir, welche Eigenschaft wir analysieren möchten, in diesem Fall die Mean-Time-To-Failure (MTTF). Über die Interaktion innerhalb des Szenarios definieren wir einen Bereich. In unserem Fall würden wir die MTTF für alle Pfade der Strategie bestimmen, die *socket->controller.plugged()* starten und mit socket->controller.unplugged() enden. Wir müssen für

jede Property, die wir oben beschrieben haben ein eigenes Schlüsselwort, also MTTF, Safety oder Reachability verwenden. Eine Programmierung eigener, neuer Properties wird dem Benutzer in SML nicht möglich sein, da diese Funktion schon in Prism exisitert. Wir müssen außerdem die Fehlerzustände identifizieren, falls sie, wie in unserem Beispiel, nicht zum terminieren des Systems in einem eigenen Fehlerendzustand führen. Dazu müssen entsprechende PA cases zusätzlich mit dem Schlüsselwort *error* gekennzeichnet werden. Diese Variante ist noch nicht optimal.

Kapitel 6

Diskussion

In diesem Kapitel diskutieren wir die in Kapitel 4 und 5 vorgestellten Lösungsansätze in Bezug auf ihre Benutzerfreundlichkeit und Implementierungsaufwand. Außerdem bewerten wir den Aufwand, eine nichtprobabilistische Spezifikation um Probabilismus zu erweitern.

6.1 Bewertung des Lösungsansatzes

Das Ziel war es probabilistische Szenarien in ScenarioTools mit SML modellieren, simulieren und analysieren zu können. Die in Kapitel 4 vorgestellte Probabilistic Alternative ist eine intuitive Variante, Wahrscheinlichkeiten in Szenarien zu modellieren. Ihre Ähnlichkeit zur bereits vorhandenen Kontrollstruktur der Alternative sollte es ScenarioTools Benutzern ermöglichen, ohne Schwierigkeiten mit ihnen umzugehen. Ein weiterer Vorteil dieser Ähnlichkeit ist, dass, obwohl die Ergebnisse dieser Arbeit noch nicht in ScenarioTools implementiert sind, wir schon jetzt Spezifikationen für die spätere Auswertung erstellen können, indem wir alle PAs durch Alternatives simulieren. Ein weiterer Vorteil dieser Ähnlichkeit ist der geringe Aufwand, der zu ihrer Implementierung betrieben werden muss. Die Änderungen an der Simulationsumgebung, um PAs darstellene zu können, sind größtenteils kosmetisch.
Um an probabilistische Spezifikationen gestellte Reliability-Fragen analysieren zu können, haben wir uns dazu entschieden einen Quellcode-Generator für Prism DTMC Modelle zu entwickeln, um die Implementierung eines Algorithmus zur probabilistischen Analyse in ScenarioTools vermeiden zu können. Prism hat bereits die benötigten Funktionen, wie eine Simulationsumgebung, die das Simulieren beliebig vieler Schritte erlaubt, und eine Auswahl an Verifikationsalgorithmen, um Properties mathematisch auswerten zu können. Allerdings bringt dieser Ansatz auch Probleme mit sich. Es wurde festgestellt, dass das generierte Prism Modell sehr unübersichtlich ist, was sich auch auf die Nutzbarkeit der Simulations-

umgebung auswirkt. Eine manuelle Zuordnung zwischen der Strategie in ScenarioTools und dem generierten Modell in Prism ist dadurch sehr schwierig. Eine Hoffnung war ursprünglich, dass Prism-kundige Benutzer mit dem generierten Modell mehr Eigenschaften analysieren können, als mit einer in ScenarioTools implementierten Variante, da sie über den kompletten Umfang der Property specification Language verfügen, welcher so nicht in ScenarioTools implementiert worden wäre. Ob das wirklich der Fall ist, oder ob es für den Benutzer dann nicht schlauer ist ein eigenes Prism Modell zu erstellen, was speziell auf die zu analysierenden Variablen ausgelegt ist, könnte man eventuell in einer weiteren Arbeit untersuchen. Für die Implementierung des, in Kapitel 5 vorgestellten, Prinzips, Properties in SML zu erstellen, sind Kenntnisse über Prism und SML nötig. Selbst nachdem dieser Property-Generator implementiert ist, muss der Benutzer immer noch ein Grundverständnis für die Arbeit mit Prism haben.

6.2 Mehraufwand der PA

Der Aufwand, probabilistische Spezifikationen zu erstellen oder bereits vorhandene um PAs zu erweitern, hängt stark von der Komplexität des zu modellierenden Systems und des gewünschten probabilistischen Einflusses ab. In dieser Arbeit haben wir z.b. nur einzelne Szenarien abgewandelt, ohne groß Einfluss auf den Verlauf des restlichen Szenarios zu nehmen. Eine Spezifikation zu erstellen, bei der mehrere PAs mit je mehreren Cases die Struktur der Spezifikation grundlegend beeinflussen, wird entsprechend aufwändiger sein. Dabei muss man immer darauf achten, bereits modellierte Anforderungen nicht zu sabotieren. Ab einem gewissen probabilistischen Anteil an der Spezifikation ist es daher ratsam, den Entwicklungsprozess von Vorne zu beginnen und sich in einem Schema zunächst über alle Abhängigkeiten klar zu werden.

Kapitel 7

Verwandte Arbeiten

In diesem Kapitel finden wir Arbeiten, die sich mit ähnlichen Problemen wie dieser Arbeit auseinandersetzen. Es existieren bereits mehrere Arbeiten zu szenariobasierter Modellierung, auf denen ScenarioTools teilweise aufbaut.

7.1 Alternativen zu ScenarioTools

Mit **PlayGo** haben Harel et al. eine auf *Live Sequence Charts*(LSC) und der Play-in/Play-out Methode basierende IDE für szenariobasiertes Entwickeln vorgestellt[5]. Der darin verwendete play-out-algorithmus ist der selbe, den auch ScenarioTools verwendet. SML basiert auf einer textuellen Variante der visuellen LSC.

7.2 Alternativen zu Prism

Der **Markov Reward Model Checker** (MRMC) ist ein Programm zur Verifizierung von Eigenschaften eines probabilistischen Modells. Es Unterstützt, wie Prism, diskrete und kontinuierliche Markovketten (DTMC & CTMC) als Modelle, und Properties in den temporalen Logiken PCTL und CSL, sowie ihrer Reward-Extensions[6].

Zusammenfassung und Ausblick

7.3 Zusammenfassung

In dieser Arbeit haben wir ein Konzept vorgestellt, mit dem probabilistische szenariobasierte Spezifikationen in ScenarioTools modelliert und dargestellt werden können. Dazu haben wir in Kapitel 4 die *Probabilistic Alternative* vorgestellt. Dabei handelt es sich um eine intuitive, auf Verzweigungen basierende, Kontrollstruktur. Damit wir die probabilistischen Erweiterungen auch darstellen können, haben wir außerdem Erweiterungen an der

ScenarioTools Simulationsumgebung sowie dem Play-out Algorithmus besprochen, und festgestellt, dass keine tiefgreifenden Änderungen nötig sind. Um die, aus unserer Spezifikation generierten Controller-Strategien probabilistisch analysieren zu können, haben wir konzeptuell einen Code-Generator entwickelt, der aus ihr ein Modell für den Prism Model Checker erzeugt. Um die, in Kapitel 3 vorgestellten, Reliability-Fragen für unsere Spezifikation beantworten zu können, stellen wir in Kapitel 5 die entsprechenden Properties vor, die Prism verifizieren soll. Dabei handelt es sich um Ausdrücke aus mehreren temporären Logiken. Anschließend zeigen wir noch eine Variante, diese Properties möglichst intuitiv in SML zu definieren.
Das Modellieren mit PAs ist sehr einfach und für ScenarioTools Benutzer sollte es, wenn, nur eine sehr kurze Lernphase geben. Wie gut die Arbeit mit Properties generell und dem insbesondere dem Generator aus SML funktioniert, können wir noch nicht abschließend bestimmen, da noch keine Implementierung vorliegt. Damit ist uns, zumindest für die Modellierung von Probabilismus in szenariobasierten Spezifikationen eine gute Lösung gelungen.

7.4 Ausblick

In dieser Arbeit wurde nur kurz auf die Möglichkeit eingegangen, den Probabilismus bereits in der Synthese von Controllern zu verwenden. Man könnte dann z.B Rewards, also probabilistische Kosten, minimieren. Auch sollte man untersuchen, ob eine Verbindung mit der kostenoptimierenden Synthese Sinn macht.
Es sollte noch abschließend, z.B mit einer Benutzerstudie untersucht werden, ob die Verwendung von Prism zur Analyse der Modelle, aus einer Usability Sichtweise, die richtige Wahl ist, oder ob doch eine direkte Implementierung in ScenarioTools gefordert wird, in der Anfragen dann möglicherweise direkt am Zustandsgraphen oder in der Strategie gestellt werden können.

Anhang A

Anhang

A.1 Inhalte der CD

A.2 Strategien

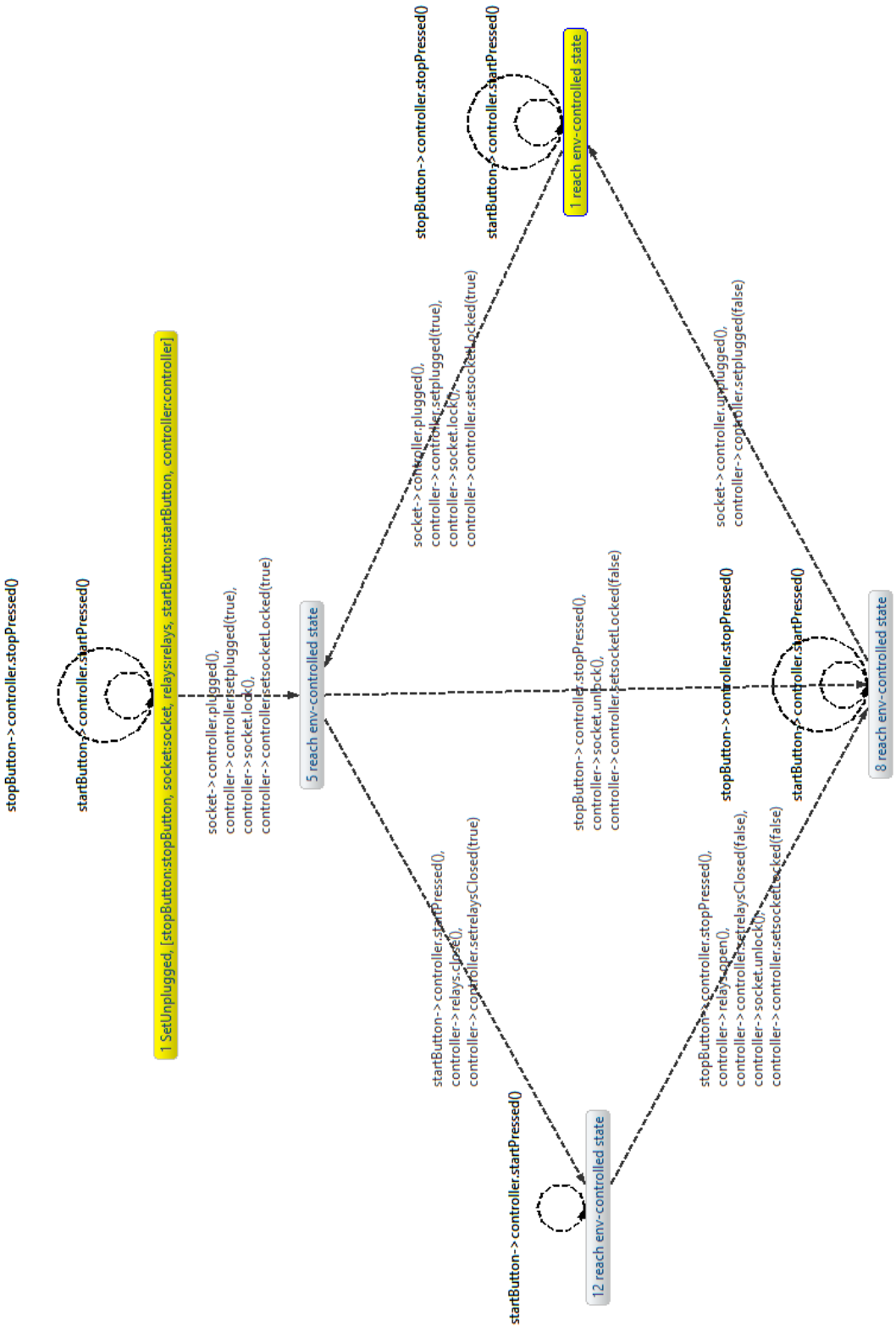

Abbildung A.1: Die Strategie der HVCS Spezifikation größer dargestellt.

A.3 Code

```
1   import "../../model/highvoltagecoupling.ecore"
2
3   specification HighvoltagecouplingSpecification {
4
5   domain highvoltagecoupling
6
7   controllable {
8   Controller
9   }
10
11  non-spontaneous events {
12  Controller.unplugged
13  }
14
15  collaboration PlugAndStart {
16
17  static role Socket socket
18  static role Relays relays
19  static role StartButton startButton
20  static role StopButton stopButton // added to allow unplugging
21  static role Controller controller
22
23
24  /*
25  * When the plug is plugged into the socket,
26  * then the socket must be locked.
27  */
28  guarantee scenario WhenPluggedThenLock{
29  socket->controller.plugged()
30  strict urgent controller->socket.lock()
31  }
32
33  guarantee scenario SetPlugged{
34  socket->controller.plugged()
35  strict urgent controller->controller.setPlugged(true)
36  }
37
38  guarantee scenario SetLocked{
39  controller->socket.lock()
40  strict urgent controller->controller.setSocketLocked(true)
41  }
42
43  /*
44  * When the start-button is pressed,
45  * then the relays have to be closed.
46  */
47  guarantee scenario WhenStartPressedThenCloseContact{
48  startButton->controller.startPressed()
49  interrupt [!controller.plugged || !controller.socketLocked || controller.
        relaysClosed]
50  strict urgent controller->relays.close()
51  }
52
53  /*
54  * When the plug is not plugged or not
55  * locked then the relays must not be closed.
56  * -> Modeled as an anti-scenario: closing the relays
57  *    leads to a violation if not plugged or not locked
58  */
59  guarantee scenario OnlyCloseRelaysWhenPlugPluggedAndLocked {
60  controller->relays.close()
61  violation [!controller.plugged || !controller.socketLocked]
62  }
63
64  guarantee scenario SetRelaysClosed{
65  controller->relays.close()
66  strict urgent controller->controller.setRelaysClosed(true)
67  }
68
69  /*
```

```
70      * When the relays are closed, the plug
71      * must not be unplugged.
72      */
73      guarantee scenario UnplugForbiddenWhenRelaysClosed {
74      socket->controller.unplugged()
75      violation [controller.relaysClosed]
76      }
77
78      assumption scenario NoUnplugBetweenLockAndUnlock{
79      controller->socket.lock()
80      controller->socket.unlock()
81      }constraints[
82      forbidden socket->controller.unplugged()
83      ]
84
85
86      /*
87      * Stop the flow of electricity to when pressing the
88      * stop button to allow unplugging
89      */
90      guarantee scenario OpenRelaysWhenStopping {
91      stopButton->controller.stopPressed()
92      interrupt [!controller.relaysClosed]
93      strict urgent controller->relays.open()
94      }
95
96      guarantee scenario ReleaseLockWhenStopping {
97      stopButton->controller.stopPressed()
98      interrupt [!controller.socketLocked]
99      strict urgent controller->socket.unlock()
100     }
101
102     guarantee scenario SetRelaysOpen {
103     controller->relays.open()
104     strict urgent controller->controller.setRelaysClosed(false)
105     }
106
107     guarantee scenario SetUnlocked {
108     controller->socket.unlock()
109     strict urgent controller->controller.setSocketLocked(false)
110     }
111
112     /*
113     * Perform unplugging
114     */
115     guarantee scenario SetUnplugged {
116     socket->controller.unplugged()
117     strict urgent controller->controller.setPlugged(false)
118     }
119
120     /*
121     * user has to unplug plug before plugging it back in
122     */
123     assumption scenario NoRepeatedPluggingIn{
124     socket->controller.plugged()
125     socket->controller.unplugged()
126     }constraints[
127     forbidden socket->controller.plugged()
128     ]
129
130     /*
131     * user eventually unplugs plug
132     */
133     assumption scenario UserUnplugsPlugAfterPressingStop {
134     socket->controller.plugged()
135     socket->controller.unplugged()
136     }
137     }
138
139     }
```

Code 11: Vollständige SML-Spezifikation des HVCS.

```
1
2  grammar org.scenariotools.sml.collaboration.Collaboration with org.scenariotools.
       sml.expressions.ScenarioExpressions
3
4  import "http://www.scenariotools.org/sml"
5  import "http://www.eclipse.org/emf/2002/Ecore"
6  import "http://www.scenariotools.org/sml/expressions/ScenarioExpressions"
7
8  Collaboration returns Collaboration:
9  imports+=Import*
10 ('domain' domains+=[EPackage|FQN])*
11 ('contexts' contexts+=[EPackage|FQN])*
12 'collaboration' name=ID '{'
13 roles+=Role*
14 scenarios+=Scenario*
15 '}';
16
17 // Terminal-Definitions
18 // ------------------------------
19 FQN:
20 ID ('.' ID)*;
21
22 Role returns Role:
23 ((static?='static') | 'dynamic' (multiRole?='multi')?) 'role' type=[EClass] name=
       ID;
24
25 // Scenario
26 // ------------------------------
27 Scenario returns Scenario:
28 singular?='singular'? kind=ScenarioKind 'scenario' name=ID
29 ((optimizeCost?='optimize' 'cost')|('cost' '[' cost=DOUBLE ']'))?
30 ('context' contexts+=[EClass] (',' contexts+=[EClass])*)?
31 ('bindings' '[' roleBindings+=RoleBindingConstraint* ']')?
32 ownedInteraction=Interaction;
33
34 enum ScenarioKind returns ScenarioKind:
35 assumption='assumption' |
36 guarantee='guarantee' |
37 existential='existential';
38
39 RoleBindingConstraint returns RoleBindingConstraint:
40 role=[Role] '=' bindingExpression=BindingExpression;
41
42 // Binding-Expressions
43 // ------------------------------
44 BindingExpression returns BindingExpression:
45 FeatureAccessBindingExpression;
46
47 FeatureAccessBindingExpression returns FeatureAccessBindingExpression:
48 featureaccess=FeatureAccess;
49
50 // Interaction
51 // ------------------------------
52 InteractionFragment returns InteractionFragment:
53 Interaction | ModalMessage | Alternative | Loop | Parallel | ConditionFragment |
       TimedConditionFragment | VariableFragment;
54
55 VariableFragment returns VariableFragment:
56 expression=(TypedVariableDeclaration | VariableAssignment | ClockDeclaration |
       ClockAssignment);
57
58 Interaction returns Interaction:
59 {Interaction}
60 '{'
61 fragments+=InteractionFragment*
62 '}' constraints=ConstraintBlock?;
63
64 enum ExpectationKind returns ExpectationKind:
65 eventually='eventually' |
66 requested='requested' |
67 urgent='urgent' |
68 committed='committed';
69
```

```
70  ModalMessage returns ModalMessage:
71  strict?='strict'? (monitored?='monitored'? expectationKind=ExpectationKind)?
        sender=[Role] '->' receiver=[Role]
72  '.'
73  (modelElement=[ETypedElement]) ('.' collectionModification=CollectionModification
        )? ('(' (parameters+=ParameterBinding
74  (','
75  parameters+=ParameterBinding)*)? ')')?;
76
77  ParameterBinding returns ParameterBinding:
78  bindingExpression=ParameterExpression;
79
80  ParameterExpression:
81  WildcardParameterExpression | ValueParameterExpression |
        VariableBindingParameterExpression;
82
83  WildcardParameterExpression returns WildcardParameterExpression:
84  {WildcardParameterExpression} '*';
85
86  ValueParameterExpression returns ValueParameterExpression:
87  value=Expression;
88
89  VariableBindingParameterExpression returns VariableBindingParameterExpression:
90  'bind' variable=VariableValue;
91
92  ProbabilisticAlternative returns ProbabilisticAlternative:
93  {probabilisticAlternative} 'probabilistic' 'alternative'
94  cases+=ProbabilisticCases ('or' cases+=ProbabilisticCases)+;
95
96  ProbabilisticCase returns ProbaibilisticCase:
97  {case}
98  '['probability = DOUBLE ';' reward = DOUBLE ']'
99  caseInteraction=Interaction;
100
101 Alternative returns Alternative:
102 {Alternative} 'alternative'
103 cases+=Case ('or' cases+=Case)*;
104
105 Case returns Case:
106 {Case}
107 (caseCondition=Condition)?
108 caseInteraction=Interaction;
109
110 Loop returns Loop:
111 'while' loopCondition=Condition?
112 bodyInteraction=Interaction;
113
114 Parallel returns Parallel:
115 {Parallel} 'parallel'
116 parallelInteraction+=Interaction ('and' parallelInteraction+=Interaction)*;
117
118 // Condition
119 // ------------------------------
120 TimedConditionFragment returns TimedConditionFragment:
121 TimedWaitCondition | TimedInterruptCondition | TimedViolationCondition;
122
123 ConditionFragment returns ConditionFragment:
124 WaitCondition | InterruptCondition | ViolationCondition;
125
126 WaitCondition returns WaitCondition:
127 'wait' strict?='strict'? requested?='eventually'? '[' conditionExpression=
        ConditionExpression ']';
128
129 InterruptCondition returns InterruptCondition:
130 'interrupt' '[' conditionExpression=ConditionExpression ']';
131
132 ViolationCondition returns ViolationCondition:
133 'violation' '[' conditionExpression=ConditionExpression ']';
134
135 TimedWaitCondition returns TimedWaitCondition:
136 'timed' 'wait' strict?='strict'? requested?='eventually'? '['
        timedConditionExpression=TimedExpression ']';
137
138 TimedViolationCondition returns TimedViolationCondition:
```

```
139 'timed' 'violation' '[' timedConditionExpression=TimedExpression ']';
140
141 TimedInterruptCondition returns TimedInterruptCondition:
142 'timed' 'interrupt' '[' timedConditionExpression=TimedExpression ']';
143
144 Condition returns Condition:
145 '[' conditionExpression=ConditionExpression ']';
146
147 ConditionExpression returns ConditionExpression:
148 expression=Expression;
149
150 // Constraints
151 // ------------------------------
152 ConstraintBlock returns ConstraintBlock:
153 {ConstraintBlock} 'constraints' '['
154 (('consider' consider+=ConstraintMessage) |
155 ('ignore' ignore+=ConstraintMessage) |
156 ('forbidden' forbidden+=ConstraintMessage) |
157 ('interrupt' interrupt+=ConstraintMessage))*
158 ']';
159
160 ConstraintMessage returns Message:
161 sender=[Role] '->' receiver=[Role] '.' modelElement=[ETypedElement] ('.'
      collectionModification=CollectionModification)? '(' (parameters+=
      ParameterBinding (','
162 parameters+=ParameterBinding)*)? ')';
```

Code 12: Komplette modifizierte Collaboration.xtext

Literaturverzeichnis

[1] Tutorial on dtmc modelling in prism model checker. `www.prismmodelchecker.org/tutorial/die.php`. [Online, Accessed 2-April-2018].

[2] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, Jul 1999.

[3] J. Greenyer, D. Gritzner, T. Gutjahr, F. König, N. Glade, A. Marron, and G. Katz. Scenariotools – a tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming*, 149:15 – 27, 2017. Special Issue on MODELS'16.

[4] D. Gritzner and J. Greenyer. Synthesis of cost-optimized controllers from scenario-based gr(1) specifications. In I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, and C. Seidl, editors, *Modellierung 2018*, pages 167–182, Bonn, 2018. Gesellschaft für Informatik e.V.

[5] D. Harel, S. Maoz, S. Szekely, and D. Barkan. Playgo: Towards a comprehensive tool for scenario based programming. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 359–360, New York, NY, USA, 2010. ACM.

[6] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, 68(2):90 – 104, 2011. Advances in Quantitative Evaluation of Systems.

[7] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[8] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In T. Field, P. G. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 200–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.