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

Testen von szenariobasierten Spezifikationen zum verbesserten Verständnis von Inkonsistenzen

Bachelorarbeit

im Studiengang Informatik

von

Michael Warnke

Prüfer: Professor Dr. Joel Greenyer
Zweitprüfer: Professor Dr. Kurt Schneider
Betreuer: Dipl. - Inf. Daniel Gritzner

Hannover, 21.12.2017

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 21.12.2017

____

Michael Warnke

Zusammenfassung

Reaktive Systeme prägen verstärkt die Modellierung von Software- Systemen. Durch die Abstraktion des Systemverhaltens können Szenario- basierte Vorgehensmodelle, die das Verhalten der Systeme auf einem mittleren Detaillierungsgrad beschreiben, erstellt werden, wodurch in einfacher Weise ein gemeinsames Verständnis der gewünschten Systemfunktionen entwickelt wird. Die Implementierung allerdings erfolgt auf Codeebene, in der die Abstraktion des Modells verloren geht. Deshalb werden Entwicklungstools benötigt, die die Lücke zwischen Entwurf und Implementierung schließen. Eines dieser Entwicklungstools ist ScenarioTools. ScenarioTools beschäftigt sich mit der Analyse und der Modellierung reaktiver Systeme. Dazu bietet ScenarioTools die Möglichkeit, Szenarien in einer domänenspezifischen Sprache zu formulieren und die Realisierbarkeit dieser zu überprüfen. Im Falle von Inkonsistenzen in der Spezifikation sind diese zu finden und zu beheben, sodass die Spezifikation wieder realisierbar ist. Um das Debugging von Inkonsistenzen zu erleichtern, wird in dieser Arbeit ein Plugin zur Erweiterung von ScenarioTools entwickelt, das die einfache Identifizierung von Inkonsistenzen ermöglicht und somit zu einem vereinfachten Arbeiten mit ScenarioTools beiträgt.

Abstract

Reactive systems increasingly have an impact in the modeling of software-intensive systems. The characteristic of reactive systems to describe system behaviour on a abstract level, eases the understandment of the desired system specifications. The resulting effect of the use of abstraction models is a gap between the model and the implementation phase within the software development. The abstraction which initially built a base of understanding turns out to get lost when implementing system functionality. Therefore the urge for development tools which close the gap between modeling and implementation is increasing. A solution is delivered by the use of ScenarioTools. ScenarioTools is a development tool which focuses on the analysis and the modeling of reactive systems. The core funtionality lies in the formulation of scenarios on a domainspecific level and to check the realizability of these specifications. In case of inconsistencies in the specification, it is necessary to identify and fix these to make the specification realiazable. This thesis enhances the functionality of ScenarioTools, debugging process by providing a plugin which implements a easy way of identifying inconsistencies to simplify the work with ScenarioTools.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 3
1.2 Problemstellung 4
1.3 Lösungsansatz 5
1.4 Gliederung 5

2 Grundlagen **7**
2.1 Eclipse 7
2.1.1 Plugins 8
2.1.2 Eclipse Modeling Framework 8
2.2 ScenarioTools 9
2.2.1 Scenario Modeling Language - SML 10
2.2.2 SML-Spezifikation 10
2.2.3 Simulation 13
2.2.4 Synthese 14

3 Konzept **17**
3.1 Anforderungen erheben 17
3.1.1 Demonstration 18
3.1.2 Anforderungen 22
3.2 Entwurf erstellen 23
3.2.1 Anforderungen zusammenfassen und klassifizieren . . . 23
3.2.2 Prototyp des GUI 24
3.3 Verwendete Verfahren 24
3.3.1 MVC-Pattern 25
3.3.2 Tiefensuche 25
3.3.3 Beschränkte Tiefensuche 26

4 Implementierung **27**
4.1 Implementierung part One: Implementieren der Anforderungen 27
4.1.1 Anforderung a): Charakterisieren der Inkonsistenzen . 28
4.1.2 Anforderung b): Anzeigen der Inkonsistenzen 28
4.1.3 Anforderung c): Sortieren der Inkonsistenzen 29

4.1.4 Anforderung d): Abspeichern und Laden der Inkonsistenzen 30
4.1.5 Anforderung e): Editiermöglichkeit der Anzeige 31
4.1.6 Anmerkungen 31
4.2 Implementierung part Two: Kopplung mit ScenarioTools . . . 31
4.2.1 Kopplung des Views 31
4.2.2 Kopplung der Funktionalität 32

5 Evaluation **35**
5.1 Evaluation des GUI 36
5.1.1 Heuristische Evaluation 36
5.1.2 ISO 9241-110 38
5.2 Umfrage 39
5.2.1 Ergebnis der Auswertung 39
5.3 Evaluation von Funktionalität und Usability 40
5.3.1 Cognitive Walkthrough 40
5.3.2 Usability 42
5.3.3 Experiment 43
5.3.4 SOLL - IST Analyse 44
5.4 Zusammenfassung der Evaluation 45

6 Abgrenzung verwandter Arbeiten **47**
6.1 Debugging 47
6.1.1 Debugging 47
6.1.2 Einordnen der Arbeit 47
6.2 Abgrenzung zu ähnlichen Arbeiten 48

7 Zusammenfassung und Ausblick **49**
7.1 Zusammenfassung 49
7.1.1 Inhaltliche Zusammenfassung 49
7.1.2 Funktionale Zusammenfassung 49
7.2 Ausblick 50

A Ein Anhang **51**

Kapitel 1

Einleitung

Seit den 50-er Jahren konnte sich die Software neben der Hardware als eigenständiger Wirtschaftszweig etablieren. Eine offizielle Trennung von Soft- und Hardware hat nach Ceruzzi[2] aber nie stattgefunden. Als Folge konkurrierender Entwicklungsstile im Bereich der Softwareentwicklung wurde der Begriff des Software Engineerings geboren, welcher nach Balzert[1] die Bereitstellung und die Verwendung von Methodiken, Mustern und Prozeduren zur professionellen Softwareentwicklung umfasst.

Eine Methodik der Softwareentwicklung ist das Divide and Conquer Verfahren. Die Bearbeitung komplexer Aufgaben kann, wie in der ersten Ausgabe des Buches Four-Dimensional Education von Fadel und Trilling[4] beschrieben, durch das Divide and Conquer Verfahren in mehrere kleinere Teilaufgaben unterteilt werden. Dadurch kann der Fokus gezielt auf komplexe Aspekte der Teilbereiche gerichtet werden. Das Divide and Conquer Verfahren wird vor allem während der Implementierung von Algorithmen verwendet.

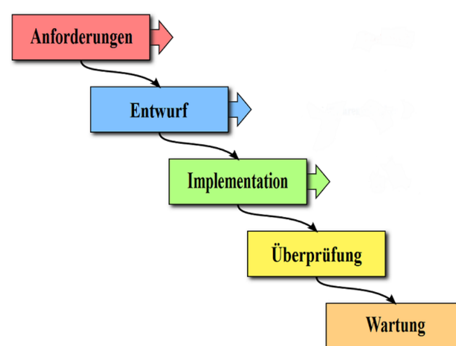

Abbildung 1.1: Wasserfallmodell nach Winston Royce

Die Implementierungsphase ist nach dem Wasserfallmodell, das in Abbildung 1.1 zu sehen ist und von Winston Royce 1970 erstmals formal

beschrieben wurde[20], die dritte Phase im Softwareentstehungsprozess. Bevor die Implementierung durchgeführt wird, werden die in der Anforderungsphase gesammelten Anforderungen an das Zielsystem in der Entwurfsphase in konkrete Entwürfe des Systems z.B. in Form von Modellen umgesetzt. Nachdem die Implementierung durchgeführt wurde, wird diese in der Testphase überprüft und anschließend im Betrieb eingeführt. Die letzte Phase der Softwareentwicklung im Rahmen des Wasserfallmodells ist die Wartung der Software. Die Wartung beinhaltet das Installieren und Bereitstellen von Updates der Software bei Erweiterung oder die Korrektur von im Nachhinein entdeckten fehlerhaften Funktionen[20].

Ähnlich wie das Divide and Conquer Prinzip Algorithmen während der Implementierung auf rudimentäre Einzelheiten reduziert, hat sich im Bereich der Entwurfserstellung die Modellierung als Sieb der Komplexität und der damit verbundenen Konzentration auf wesentliche Aspekte als standardisiertes Verfahren der Softwareentwicklung etabliert.

Durch den vielseitigen Einsatzbereich haben sich viele verschiedene Modellierungssysteme durchgesetzt. Zum Beispiel unterscheidet man zwischen statischen und dynamischen Systemen. Statische Systeme sind unabhängig von ihrer Umgebung. Sie haben keine zeitliche Beschränkung und reagieren bei gegebenem Input identisch. Dynamische Systeme hingegen verändern sich während der Ausführung. Genauer gesagt verändern sie, wie in dem Artikel über ScenarioTools von Greenyer, Gritzner, Glade, Katz, König und Marron [7] beschrieben, ihre Systemstruktur. Diese Veränderung wiederum impliziert das Verändern von Bindungen und Rollen der Systemkomponenten. Dieser Vorgang wiederholt sich bei Interaktion mit neuen Attributen, sodass dynamische Systeme eine Vielzahl von Systemzuständen produzieren können. Eine Kaffeemaschine z.B. wird durch die Betätigung von Knöpfen in einen Zustand versetzt, in dem es nicht mehr möglich ist, die Art des Kaffees zu verändern. Dadurch sind Cappuccino- und Milchkaffeeknopf ohne Funktion, obwohl sie dies zu Beginn der Interaktion noch waren. Die Interaktion bis zum aktuellen Zustand hat die Beziehungen und die Rollen der Komponenten also so geändert, dass Cappuccino- und Milchkaffeeknopf keine Reaktion des Systems mehr auslösen.

Aufgrund der Charakteristik, sich durch Reaktionen zu verändern, werden dynamische Systeme auch reaktive Systeme genannt. In vielen Domänen wie z.B. der Produktion oder der Logistik, aber auch in der Informatik spielen reaktive Systeme eine zentrale Rolle, um vor allem Szenario-basiertes Verhalten zu modellieren[7]. Szenario-basierte Systeme beschreiben das Systemverhalten auf einer mittleren Detailebene durch Vorgehensmodelle. Durch die aufeinander aufbauenden Vorgehensschritte und die charakteristische reaktive Modellierungsart werden Szenario-basierte Systeme, wie auch in Harel und Marellys Veröffentlichungen[9] zu lesen, in den meisten Fällen umgangssprachlich beschrieben. Die eigentliche Software-Implementierung wird versteckt und tritt nur selten in Erscheinung.

Viele heutzutage verwendete Software-basierte Anwendungen werden in der Entwicklung durch reaktive Systeme beschrieben. Bei der Entstehung dieser Anwendungen und der Beschreibung ihrer reaktiven Systeme werden oftmals Entwicklungstools verwendet. Entwicklungstools unterstützen die Entwicklung durch das Bereitstellen von Entwicklungsumgebungen. Die Umgebungen wiederum enthalten Funktionalitäten, die z.B. bei der textuellen Formulierung von Anforderungen oder bei der visuellen Umsetzung des Systemverhaltens Hilfestellungen geben. Eines dieser Entwicklungstools ist ScenarioTools. ScenarioTools ist ein Eclipse-basiertes Entwicklungstool und konzentriert sich[7] auf die Konstruktion, Analyse und Modellierung von Szenario-basierten reaktiven Systemen.

1.1 Motivation

Wenn ein System entwickelt werden soll und die Spezifikation gegeben ist, besteht die nächste Phase im Softwareengineeringprozess aus der Erstellung von Entwürfen, die das spezifizierte Verhalten auf einer abstrakten Basis beschreiben.[20] Besonders dann, wenn es sich um reaktive Systeme handelt und vorgegeben wird, dass sich Anforderungen im Nachhinein noch ändern können, ist es von entscheidender Bedeutung, die Interaktion der Systemkomponenten genau zu definieren.[7] Eine einzige nachträgliche Bedingung kann, wenn die Abhängigkeiten ihrer Implementierung nicht korrekt beschrieben werden, nach Harel und Marelly [9] ein funktionierendes System zerstören.

Allerdings ist der Handlungsspielraum reaktiver Systeme begrenzt und nicht alle Komponenten nehmen zentrale Rollen ein, die einem stetigen Wandel unterliegen. Betrachten wir beispielsweise das Szenario einer Kaffeemaschine. Zu Beginn kann die Kaffeemaschine per Knopfdruck Kaffee ausschenken. Nun wird das Szenario erweitert, indem die Kaffeemaschine nur dann Kaffee ausschenken kann, wenn sie eingeschaltet ist. Dem Szenario wird also eine Bedingung hinzugefügt, die das Szenario in einen neuen Zustand versetzt. Dieser Zustand wird weiter verändert, indem das Szenario erneut um die folgenden Bedingungen erweitert wird. Der Kaffee darf nur noch ausgeschenkt werden, wenn eine Kaffeetasse bereit steht und die Kaffeemaschine kann nur dann eingeschaltet werden, wenn sie an eine Stromquelle angeschlossen ist. Zusammengefasst wurde das Szenario nun um drei Bedingungen erweitert, die die Komponentenanzahl und die Abhängigkeiten der Komponenten jedes Mal erweitert und verändert haben. Durch jede der Erweiterungen wurde eine neue Anforderung an das System gestellt, die zum Teil Ähnlichkeiten aufweisen. Die erste und die zweite Anforderungen z.B. limitieren die Ausgabe des Kaffees und können dadurch gruppiert werden, wodurch die Anzahl an 'Anforderungstypen' reduziert und eine Verringerung der Komplexität, wie es auf den Seiten 80 und

folgenden des Buches 'Requirements Engineering' von Hull, Jackson und Dick steht[3], erreicht wird. Die dritte Anforderungen hingegen bildet die Problematik der Stromversorgung ab und stellt somit eine andere Gruppe von Anforderungen an das System dar. Das Erkennen von Ähnlichkeiten in den Anforderungen ist in diesem Beispiel verhältnismäßig einfach. Das liegt daran, dass es sich nur um wenige Anforderungen handelt. Arbeitet man aber mit größeren Systemen, ist es oftmals nicht so leicht, Anforderungen zu erkennen und Gruppierungen durchzuführen. Vor allem dann nicht, wenn es sich um reaktive Systeme handelt[7].

1.2 Problemstellung

Wie im vorigen Kapitel angesprochen, kann durch eine minimale Änderung der Spezifikation eine Lawine von Inkonsistenzen ausgelöst werden[9]. Inkonsistent in diesem Zusammenhang bedeutet, dass die Anforderungen der Spezifikation nicht erfüllt werden. Eine Anwendung zur Behandlung dieser Problematik ist das Entwicklungstool ScenarioTools. ScenarioTools bietet die Möglichkeit, reaktive Systeme auf einer textuellen Basis zu formulieren und sie in Form eines Zustandsgraphen zu simulieren. Weiterhin können diese Zustandsgraphen auf Erfüllung ihrer Anforderungen überprüft werden. Eine ausführliche Anleitung dazu lässt sich in dem Artikel von Greenyer, Gritzner, Glade, König, Marron und Katz finden.[7]

Die Überprüfung der Zustandsgraphen zeigt, warum die Spezifikation nicht erfüllt werden konnte. Sie kann aufzeigen, welche Szenarien während der Modellierung verletzt wurden[7]. Die Überprüfung zeigt aber nicht, ob diese Inkonsistenz schon einmal in Erscheinung getreten ist oder ob es sich um eine neue Art von Inkonsistenz handelt. Dieses Problem tritt besonders dann auf, wenn inkrementell gearbeitet wird. Dazu wird im folgenden anhand eines Beispiels gezeigt, wie dieses Problem zu Negativauswirkungen während der Entwicklung führen kann.

In einem Unternehmen wird mit ScenarioTools entwickelt. Nach dem Release eines Inkrements erhält das Team neue Anforderungen des Projektleiters. Nachdem der zuständige Ingenieur die neuen Anforderungen der Spezifikation hinzugefügt hat, führt er eine Simulation der Spezifikation durch. Die Simulation verläuft nicht ohne das Auftreten von Inkonsistenzen, und der Ingenieur verbringt Zeit damit, die Inkonsistenzen einzuordnen und zu verstehen, bevor er die Spezifikation erneut simulieren kann. Auch kann er die wiederkehrenden Inkonsistenzen nicht erkennen, weil er seine Priorität auf die Implementierung gelegt hat. Daher verbringt er wieder Zeit mit dem Verstehen und der Einordnung der Inkonsistenzen.

1.3 Lösungsansatz

Könnte der Ingenieur sich nur auf die eigentliche Implementierung konzentrieren, würde die Entwicklung effizienter durchgeführt werden und das Unternehmen Zeit und Kosten sparen. Daher ist es von Nutzen, die Identifizierung, die Gruppierung und die Dokumentation der Inkonsistenzen zu automatisieren und für den Ingenieur bereit zu stellen. Durch solch eine Erweiterung könnte der Ingenieur bei dem Implementieren neuer Anforderungen auftretende Inkonsistenzen schneller einordnen und unterscheiden, ob es sich um neue oder um wiederkehrende Inkonsistenzen handelt.

Bezogen auf ScenarioTools liegt die Lösung des oben beschriebenen Problems in der Entwicklung eines Plugins, das Inkonsistenzen übersichtlich anzeigen, charakterisieren, abspeichern und laden kann, um eine Übersicht über die vorherige und die aktuelle Spezifikationsverletzungen zu erhalten und bei inkrementeller Arbeit die Folgen von Spezifikationsänderungen schneller erkennen zu können.

1.4 Gliederung

Die Arbeit besteht aus sieben Kapiteln. Nachdem im Einführungskapitel der Arbeitsbereich und der Hintergrund der Arbeit erläutert wird, wird in Problemstellung und Lösungsansatz beschrieben, welche Thematik die Arbeit behandelt. Im zweiten Kapitel werden Grundlagen vermittelt, die nicht Teil des regulären Studiums waren, um die Funktionalität der verwendeten Tools Eclipse und ScenarioTools zu verstehen. Im dritten Kapitel wird das Konzept vorgestellt. Das Konzept ist in drei Teile aufgeteilt. Im ersten Teil wird die Problematik konkretisiert und Anforderungen an das Plugin formuliert. Im zweiten Teil werden diese klassifiziert und ein Prototyp des Graphical User Interfaces(GUI) entworfen. Der dritte Teil enthält allgemeingültige Algorithmen, Methoden und Verfahren, die verwendet wurden, um das Plugin zu entwickeln. Im vierten Kapitel wird die konkrete Implementierung des Plugins für ScenarioTools beschrieben. Dabei gliedert sich die Implementierung in zwei Teile. Im ersten Teil wird die Entstehung des Plugins erläutert. Dazu wird das Plugin zuerst ohne Anbindung an ScenarioTools entwickelt. Erst im zweiten Teil findet die Kopplung des Plugins mit ScenarioTools statt. Das fünfte Kapitel behandelt die Evaluation des Plugins. Die Evaluation gliedert sich in zwei Teile. Im ersten Teil wird die Evaluation des GUI anhand von allgemein anerkannten Prinzipien durchgeführt. Im zweiten Teil werden Funktionalität und Usabilty mit Hilfe von Evaluationstechniken untersucht. Im sechsten Kapitel wird eine Abgrenzung zu anderen Arbeiten vorgenommen. Dafür wird die Arbeit zunächst in übergeordnete Bereiche des Software-Engineerings eingeordnet, um dann ähnliche Arbeiten aufzuzeigen und letztendlich eine Abgrenzung

zu diesen durchzuführen. Abschließend besteht das siebte Kapitel aus einer Zusammenfassung, welche die wichtigsten Punkte der Arbeit aufgezeigt sowie einem Ausblick, auf die zukünftige Verwendung des Plugins.

Kapitel 2

Grundlagen

In diesem Kapitel werden Grundlagen beschrieben, die nicht zu den regulären Themen des Studiums gehören, aber wichtig sind, um die Entwicklung, Einbindung und Funktionsweise des Plugins zu verstehen. Dazu ist das Grundlagen-Kapitel in drei Abschnitte unterteilt. Im ersten Teil wird auf die Entwicklungsplattform Eclipse eingegangen. Dabei werden zuerst allgemeine Funktionen von Eclipse und danach Plugins im Zusammenhang mit ScenarioTools und dem Entwerfen eigener IDEs erklärt. Abschließend wird das Eclipse Modeling Framework vorgestellt, welches in ScenarioTools verwendet wird, um Klassenbeziehungen ähnlich wie in UML modellieren zu können[23]. Im zweiten Abschnitt wird dann näher auf ScenarioTools eingegangen. Im Fokus steht dabei, alle für die Erstellung von Spezifikation und Durchführung der Simulation wichtigen Komponenten zu beschreiben. Die Unterpunkte des ScenarioTools Abschnittes sind Scenario Modeling Lanugage(SML), die Sprache, mit der in ScenarioTools entwickelt wird, ihre Inhalte wie die Spezifikation, dem Zustandsgraphen und seine Komponenten und die Play-Out-Simulation[7]. Im dritten Abschnitt werden Verfahren vorgestellt, die für die Entwicklung des Plugins verwendet wurden. Neben der Beschreibung des MVC- Designpatterns, wird der Algorithmus 'Tiefensuche' erklärt.

2.1 Eclipse

Eclipse ist im Bereich der integrierten Entwicklungsumgebung eine weit verbreitete Anwendung. Es stellt drei Funktionsbereiche zur Verfügung, welche sie zu einer attraktiven Entwicklungsumgebung machen. Erstens die integrierte Entwicklungsumgebung(kurz: IDE), mit der Java-Programme entwickelt werden. Ein übersichtliches Framework, eine gute Repository-Abteilung, ein einfach zu bedienender Projektmanager und weitere Built-In Funktionen wie z.B. Auto-Complete ermöglichen eine angenehme Java-basierte Software-Entwicklung. Zweitens können durch die modulare Ar-

chitektur von Eclipse einzelne Plugins entfernt oder hinzugefügt werden, wodurch Eclipse die Möglichkeit bietet, IDEs zu erweitern oder eigene IDEs entwerfen zu können. Mehr dazu im Unterpunkt 'Plugins'[23]. Drittens bietet Eclipse mit dem integrierten Marketplace die Möglichkeit, IDEs durch Tools zu erweitern. Der Eclipse- Markplatz bietet eine Fülle von Tools diverser Herstellern und enthält neben funktionalen Erweiterungen auch Tools für die Modellierung, wie z.B. das Eclipse Modeling Framework(kurz: EMF), welches im Unterabschnitt 'EMF' erläutert wird[12].

2.1.1 Plugins

Wie schon im oberen Abschnitt beschrieben, bietet Eclipse die Möglichkeit, durch das Hinzufügen oder Entfernen einzelner Plugins eigene IDEs zu konfigurieren. Es gibt dabei zwei Arten von Plugins wie sie im Tutorial von vogella[23] beschrieben werden. Einmal solche, die die Unterstützung weiterer Programmiersprachen ermöglichen. Und zum anderen gibt es Plugins, die die Funktionalität des Frameworks erweitern. Die Eclipse Run Configuration erstellt eine neue Eclipse Instanz, in der die ausgewählten Plugins eingebaut werden. So können mit demselben Workplace mehrere unterschiedlichen Eclipse Instanzen durch eine gezielte Auswahl von Plugins erstellt werden.

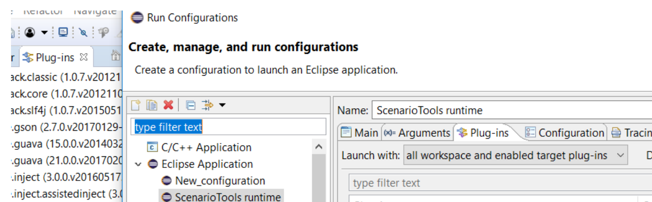

Abbildung 2.1: Laufzeitkonfiguration und Auswahl von Plug-Ins in Eclipse

Abbildung 2.1 zeigt, wie man eine Eclipse Laufzeit konfiguriert. Links unter dem Reiter Plugins sind die Plugins aufgelistet, die für die Laufzeitkonfiguration ausgewählt werden können. In der mittleren Spalte wird angegeben, welche Anwendung gestartet werden soll. Die Auswahl 'launch with:', gibt an, welche Plugins zur Erstellung der Laufzeitkonfiguration benutzt werden sollen.

2.1.2 Eclipse Modeling Framework

Das EMF ist eine Erweiterung für Eclipse. Mithilfe von EMF können Meta-Modelle erstellt werden. Meta-Modelle sind Modelle zur Modellierung inhaltlich höheren Ebenen[12]. Ein wichtiger Bestandteil des EMF sind eCore- Modelle. eCore- Modelle sind Domänenmodelle zur Beschreibung von Komponenten und ihrer Attribute. Sie können im EMF-Tree-Editor angesehen und editiert werden. Weiterhin können sie, wie es detailliert

von Koegel und Helmig [12] beschrieben wird, mit dem XML-Editor verändert und mithilfe von UML-Style modelliert werden. Ähnlich wie UML-Diagramme beschreiben eCore- Modelle ihre Klassen und die Beziehungen untereinander. Weiterhin können eCore-Modelle in Java-Code umgewandelt werden, wodurch das EMF die drei Bereiche Java, UML und XML kombiniert.

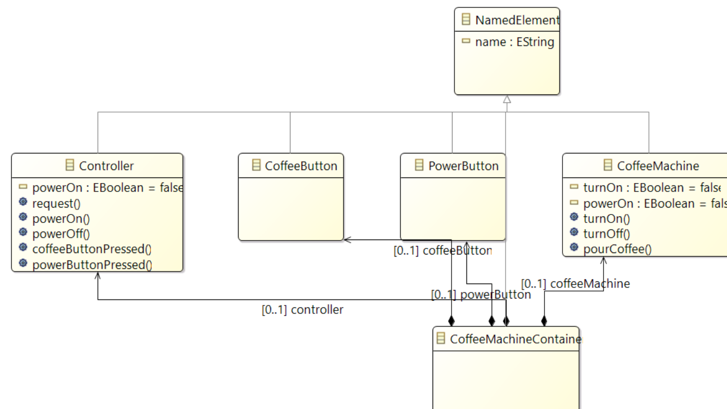

Abbildung 2.2: eCore Klassendiagramm

In Abbildung 2.2 ist das eCore-Klassendiagramm eines minimal ausgeführten Kaffeeautomaten zu sehen. Wie in der UML werden Objekte bei eCore in Form von Rechtecken mit ihrem Namen im oberen Teil beschrieben. Attribute, wie die boolean-Werte der CoffeeMachine, werden durch Rechtecke angezeigt und Methoden durch blaue Kreise notiert. Kardinalitäten zwischen Objekten werden durch eine Nummerierung an der Beziehungskante definiert, und die Raute gibt Vererbungsrichtungen zwischen zwei Klassen an.

2.2 ScenarioTools

Die Modellierung von reaktiven Systemen ist in vielen Domänen zu einem eigenständigen Entwicklungszweig geworden. ScenarioTools entstand in einem akademischen Projekt der Leibniz Universität Hannover, das sich mit der Analyse und der Modellierung dieser Systeme beschäftigt. Basierend auf Eclipse ist ScenarioTools eine Ansammlung von Plugins, die im Zusammenspiel mit dem EMF die Möglichkeit bietet, Systeme in der Domain Specific Language(DSL) Scenario Modeling Language(SML) textuell zu formulieren und ihr Verhalten durch einen Zustandsgraph zu simulieren. Domain Specific Languages sind Sprachen, die, wie in der Bachelorarbeit von Kadioglu erklärt[11], speziell für die Formulierung eines Bereiches entworfen wurden. Dabei verwendet ScenarioTools die SML, um

das Szenario-basierte Verhalten der Systeme, textuell zu formulieren. Die Bestandteile der SML sind, wie auf der Website ScenarioTools nachzulesen[6], darauf zugeschnitten, die Interaktion zwischen Systemkomponenten im Detail zu beschreiben. Durch die Simulation des textuell formulierten Verhaltens ist ScenarioTools in der Lage, das Systemverhalten in Form eines Zustandgraphen zu visualisieren.

2.2.1 Scenario Modeling Language - SML

SML ist eine DSL, die entwickelt wurde, um die Interaktion von Systemkomponenten textuell beschreiben zu können. Sie fungiert als Übergang von den zunehmend verwendeten Vorgangsmodellen zur Beschreibung des Systemverhaltens in die Formulierung des Verhaltens auf Codeebene. Sie ist die Sprache, mit der in ScenarioTools gearbeitet wird und basiert auf den Live Sequence Charts, welche unter anderem von Harel und Marelly [9] genauer erklärt werden. Der Vorteil in der Verwendung von SML liegt in der Art und Weise, wie sie die Interaktion der Systemkomponenten festlegt. Im folgenden Kapitel werden die für diese Arbeit wichtigen Komponenten der Sprache SML anhand eines Beispiels, welches das Verhalten einer Kaffeemaschine enthält, beschrieben.

2.2.2 SML-Spezifikation

Die SML-Spezifikation ist in mehrere Abschnitte unterteilt. Neben dem Import von eCore-Klassendiagrammen der beteiligten Systeme und der Deklaration der Komponenten in System- und Umweltvariablen, enthält die Spezifikation die in SML formulierte Interaktion der Systemkomponenten. Im Folgenden wird die Erstellung einer Spezifikation in ScenarioTools demonstriert.

```
import "../model/CoffeeMachine.ecore"

specification CoffeeMachineSpecification {

    domain CoffeeMachine

    controllable{
        Controller
    }
```

Abbildung 2.3: Imports und Rollen der SML- Spezifikation

Wie in Abbildung 2.3 zu sehen, wird in der ersten Zeile die eCore- Datei des Kaffeemaschine-Systems importiert. Dadurch können alle Komponenten des Kaffeemaschinen-Systems in dieser Spezifikation verwendet werden.

Anschließend wird die Spezifikation initialisiert. Der erste Schritt nach dem Initialisieren der Spezifikation besteht darin, eine domain zu wählen.

Durch die Wahl der Domain, können die Komponenten des gewählten eCore-Modells als *controllable* definiert werden. Die Definition aller controllables ist gleichzeitig auch der nächste Schritt in der Spezifikation. Wenn eine Komponente Teil des Systems ist, erhält sie das Attribut *'controllable'*, wodurch gekennzeichnet wird, dass ihr Verhalten spezifiziert werden soll[6]. Umweltdomains hingegen sind passive, *'uncontrollable'* Akteure, welche auf Aktionen der kontrollierbaren Akteure reagieren. Die Beschreibung der Spezifikation wird ausführlich in dem Aritkel über ScenarioTools(2017)[7]erklärt.

Collaboration

Die Collaboration enthält die formale Beschreibung der Interaktion der Systemkomponenten in der SML.[6].

```
collaboration CoffeeMachine {

    static role Controller      controller
    static role CoffeeMachine   coffeeMachine
    static role PowerButton     powerButton
    static role CoffeeButton    coffeeButton

    guarantee scenario turnOn{
        powerButton -> controller.powerButtonPressed()
        interrupt[controller.powerOn]
        strict urgent controller -> coffeeMachine.turnOn()
        strict urgent controller -> controller.setPowerOn(true)
    }
```

Abbildung 2.4: Rollendefinition und erstes Scenario in SML

In Abbildung 2.4 ist der Auszug einer Collaboration zu sehen. Die Collaboration beginnt damit, alle Objekte, deren Verhalten spezifiziert werden soll, aufzuzählen und ihnen eine Rolle zuzuweisen. Die Rollenzuweisung gibt an, wie sich Komponenten im Zusammenspiel mit den anderen Komponenten verhalten. Eine Komponente kann entweder dynamisch oder statisch sein. Ist eine Komponente dynamisch, bedeutet dies, dass sie an jede Komponente, die während der Ausführung kompatibel ist, gebunden werden kann. Statische Komponenten hingegen sind während der Ausführung zu genau einer anderen Komponente gebunden[6].

```
import "CoffeeMachine.sml"
configure specification CoffeeMachineSpecification
use instancemodel "CoffeeMachine.xmi"

rolebindings for collaboration CoffeeMachine{

    object CoffeeMachine.controller plays role controller
    object CoffeeMachine.powerButton plays role powerButton
    object CoffeeMachine.coffeeButton plays role coffeeButton
    object CoffeeMachine.coffeeMachine plays role coffeeMachine
```

Abbildung 2.5: Runconfig Datei der Kaffeemaschine

Zusätzlich zu der Deklaration in der Collaboration müssen statische Zuweisungen in einer sogenannten Runconfig-Datei, wie sie in Abbildung 2.5 zu sehen ist, notiert werden. Die Runcofig-Datei enthält die Bindings der Komponenten zur Laufzeit. Nach den Rollenzuweisungen beginnt die Formulierung der Szenarien[6].

Ein Szenario besteht, wie in Abbildung 2.4 zu sehen, aus Deklaration und Interaktionsteil. Die Deklaration enthält den Typ des Scenarios und den Namen. Ein Scenario kann dabei zwei unterschiedliche Typen annehmen. Entweder ist es vom Typ *'assumption'* oder vom Typ *'guarantee'*. Die Typen beschreiben die Art und Weise, wie das Scenario die Interaktion zwischen Akteur und Umwelt beschreibt[7]. Das im Beispiel mit *'guarantee'* versehene Scenario sagt aus, dass die im Scenario auftretenden kontrollierbaren Komponenten mit der Umwelt agieren dürfen oder müssen. In *'assumption'* Scenarios wird beschrieben, wie die Umwelt auf Ereignisse reagiert. Die Nachrichten repräsentieren die Kommunikation der Komponenten. Dabei können sie im Fall von Aktionsnachrichten den Austausch zwischen zwei oder mehreren Komponenten beschreiben. Sind sie Umgebungsnachrichten, enthalten Sie die Reaktion der Umwelt auf Aktionen der kontrollierbaren Komponenten. ScenarioTools stellt die Möglichkeit bereit, Nachrichten mit Modalitäten zu versehen. Modalitäten modifizieren Nachrichten, sodass die Ausführung dieser unter anderen Bedingungen durchgeführt wird[7].

Wie in Abbildung 2.4 zu sehen, werden die Nachrichten des *turnOn()*-Scenarios mit den Modalitäten *'strict'* und *'urgent'* versehen. Die *'strict'*-Anweisung fügt der Ausführung der Nachricht die Bedingung hinzu, dass sie genau an dieser Stelle in der Ausführung des Scenarios auftreten muss. Wird z.B. der interne Wert der Kaffeemaschine auf 'an' geschaltet, bevor die Kaffeemaschine eingeschaltet wurde, wird die *'strict'* Anweisung der Nachricht verletzt und eine *safety*- violation, die in den folgenden Absätzen erklärt wird, produziert. *'Urgent'* impliziert die Bedingung, dass die Nachricht sofort ausgeführt werden soll. Das bedeutet, dass keine Nachricht eines anderen Scenarios vor der Ausführung der *'urgent'*- Nachricht ausgeführt werden darf. Die nicht aufgeführte *'requested'*- Anweisung verlangt, dass die Nachricht während der Simulation durchgeführt wird. Wird die Nachricht nicht ausgeführt, so wird eine *liveness*- violation produziert.

Um das Zusammenspiel der Scenarios zu überwachen, gibt es neben den Modalitäten die sogenannten Bedingungsabfragen, welche die Ausführung von Nachrichten unterbinden kann. Die *'violation'*-Kondition gibt einen Fehler aus, wenn ihre Bedingung nicht erfüllt wird. Eine *'interrupt'*-Kondition, wie sie in Abbildung 2.4 vorkommt, unterbricht die Ereigniskette, sobald ihre Abfrage eintritt.

Die hier beschriebenen Funktionen von ScenarioTools sind nur ein für diese Arbeit wichtiger Auszug der Funktionsbreite von ScenarioTools. In dem Artikel von Greenyer, Gritzner, Glade, Katz, König und Marron[7] werden hier aufgeführte und weitere Funktionen detailliert erläutert.

Inkonsistenzen

Inkonsistenzen bedeuten die Nichterfüllung von in der Spezifikation verlangtem Verhalten. Dabei können Inkonsistenzen, wie sie in [7] beschrieben werden, zwei Formen annehmen. Einer der Typen einer Inkonsistenz ist die *safety*-violation. Eine *safety*-violation tritt auf, wenn ein Scenario, wie oben beschrieben, durch die Reihenfolge seiner Nachrichten verletzt wird.

Der andere Typ der Inkonsistenzen ist die *liveness*-violation. Eine *liveness*-violation tritt z.B. auf, wenn eine *requested*-Nachricht aktiv ist, aber nicht umgesetzt wird. Dies kann vorkommen, wenn die aktiven Scenarios einen Zyklus bilden, durch den die *requested* Nachricht nicht ausgeführt werden kann.

2.2.3 Simulation

Die Simulation erstellt aus der SML-Spezifikation einen Zustandsgraphen. Dabei wird jedes Scenario mithilfe des Play-Out-Algorithmus, der im folgenden erklärt wird, in eine Menge von Transitionen und Zuständen umgesetzt.

Algorithm 1 Play-out event selection and execution

1: Σ_{Sys} = System message events that correspond to enabled requested messages in guarantee scenarios
2: **if** $\Sigma_{Sys} \neq \emptyset$ **then** // system step:
3: $\Sigma_{ex} = \Sigma_{Sys} \setminus$ Message events that are blocked by an active guarantee scenario or disallowed by a corresponding GTR.
4: **if** $\Sigma_{ex} \neq \emptyset$ **then**
5: σ_{ex} = *selectFrom*(Σ_{ex}); *perfromStep*(σ_{ex}); **goto** 1
6: **else** // Safety-violation occurred in guarnatees
7: **goto** 9 // Cont. env. steps, check for subseq. assumption violation.
8: **else**// environment step:
9: Σ_{Env} = Environment message events that are spontaneous or correspond to enabled messages in assumption scenarios.
10: $\Sigma_{ex} = \Sigma_{Env} \setminus$ Message events that are blocked by an active assumption scenario or disallowed by a corresponding GTR.
11: **if** $\Sigma_{ex} \neq \emptyset$ **then**
12: σ_{ex} = *selectFrom*(Σ_{ex}); *perfromStep*(σ_{ex}); **goto** 1
13: **else**
14: *terminate("Assumption violation occurred")*

Abbildung 2.6: Pseudocode des PlayOut-Algorithmus

In Abbildung 2.6 ist der Pseudocode des Play-Out-Algorithmus abgebildet[7]. Der Algorithmus startet in einem Zustand, in dem keine Scenarios aktiv sind. Zu Beginn überprüft er, ob Systemnachrichten vorliegen, die mit aktiven requested Nachrichten übereinstimmen. Ist dies nicht der Fall, so handelt es sich um Umweltnachrichten und der Algorithmus fährt in Zeile acht fort. Lassen sich welche finden, filtert der Algorithmus diese aufgrund von anderen aktiven guarantee Scenarios auf geblockte Nachrichten oder auf durch eine zugehörige Graph-Transition-Rule(GTR)[7] nicht erlaubte Nachricht. GTRs können in bestehende Zustandsgraphen

eingefügt werden, um Bedingungen bei der Ausführung von Nachrichten hinzuzufügen[7]. Wenn nach dem Filtern immer noch Systemnachrichten vorhanden sind, führt der Algorithmus die *performStep(MessageEvent)*-Methode aus, welche eine Transition, die aus der Systemnachricht besteht, erstellt und einen Folgezustand generiert. Existieren nach dem Filtern geblockter Nachrichten für die *guarantee* Scenarios keine Systemnachrichten mehr, handelt es sich um eine *safety*-violation, und der Algorithmus springt in Zeile 9, weil die übrig gebliebenen aktiven Scenarios vom Typ *assumption* sind. Die Prozedur für die assumption-Scenarios ist dieselbe wie die für die guarantee-Scenarios. Wenn allerdings nach dem Filtern weder System- noch Umweltnachrichten vorliegen, so springt der Algorithmus in Zeile 14 und terminiert aufgrund der Verletzung einer Umweltnachricht. Eine Beschreibung des Algorithmus lässt sich in [7] finden.

So iteriert der Play-Out-Algorithmus nicht-deterministisch über alle Ereignisse aktiver Scenarios und konstruiert schrittweise alle Zustände und Transitionen.

Zu beachten ist, dass der Algorithmus bei der Verletzung von *assumption*-Scenarios direkt terminiert, aber bei der Verletzung von *guarantee*-Scenarios weiterläuft, um zu überprüfen, ob die Umwelt eine guarantee Verletzung auf Kosten einer später auftretenden *assumption*- Verletzung in Kauf nimmt.

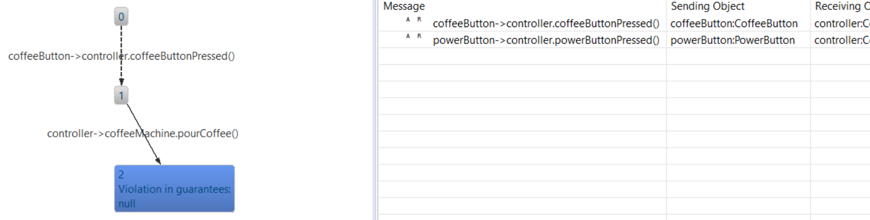

Abbildung 2.7: Event Selection View von ScenarioTools

Nachdem der Play-Out-Algorithmus den Zustandsgraphen konstruiert hat, kann der Ingenieur die Simulation wie in Abbildung 2.7 im *Message Event Selection View* durchführen. Durch das Auswählen der Nachrichten kann er diese nacheinander modellieren lassen und überprüfen, ob durch den aktuellen Schritt eine *violation* hervorgerufen wird[6].

2.2.4 Synthese

Bei dem Auftreten von Inkonsistenzen während der Simulation ist ScenarioTools in der Lage, eine Synthese des Zustandsgraphen durchzuführen. Dazu bietet ScenarioTools, wie in [7] beschrieben, die Möglichkeit, eine Counter-Strategie zu durchlaufen, die überprüft, ob die Inkonsistenz umgangen werden kann und die SML-Spezifikation realisierbar ist. Der sogenannte Realizability-Checking Algorithmus von ScenarioTools erstellt dazu einen

Zustandsgraphen aller Play-Out Ausführungen. Er sieht die Konstruktion des Play-Out-Graphen als ein Spiel zwischen System und Umwelt, in dem es darum geht, eine Strategie zu finden, in der es mindestens einen System-kontrollierten Pfad gibt, der bei jeglicher Auswahl von Umweltereignissen die SML-Spezifikation erfüllen kann. Das Spiel gilt als gewonnen, wenn es solch eine Strategie gibt. Im Fall einer Niederlage zeigt der Realizabilty-Algorithmus durch eine Counter-Strategie an, wie die Umwelt eine Verletzung erzwingen kann.

Kapitel 3

Konzept

Das Konzept beschreibt die Erstellung eines Entwurfes für die Implementierung und ist somit eines der zentralen Kapitel der Arbeit. Der Entwurf enthält die konzeptuelle Herangehensweise an das Lösen der Problematik und die Entwicklung des Plugins. Dazu gliedert sich das Konzept in drei Teile. Im ersten Teil wird im Rahmen des Requirement-Engineering die Anforderungserhebung durchgeführt. Damit stellt dieser Abschnitt bezogen auf das Wasserfallmodell aus Abbildung 1.1 den Start der Entwicklung dar. Anhand einer Demonstration des Tools am Beispiel der Kaffeemaschine wird gezeigt, wie die Entwicklung mit ScenarioTools verbessert werden kann. Weiterhin werden, basierend auf den Beobachtungen der Demonstration, Anforderungen an das zu entwickelnde Plugin erhoben. Im zweiten Teil findet gemäß des Wasserfallmodells die Entwurfsphase statt. Dazu werden die im ersten Teil erhobenen Anforderungen analysiert und prototypisch dargestellt, wie das Plugin aussehen könnte. Im dritten Teil des Konzepts werden Verfahren beschrieben, die verwendet werden, um das Plugin zu entwickeln. Eines dieses Verfahren ist die Anwendung eines MVC- Designpatterns. Ein weiteres Verfahren ist der Tiefensuche- Algorithmus, der verwendet wird, um Baumstrukturen in linearer Laufzeit zu durchsuchen.

3.1 Anforderungen erheben

Um ein besseres Verständnis der Problematik zu erhalten, wird in diesem Unterabschnitt am Beispiel einer Kaffeemaschine mit minimalistischer Funktionalität die Entstehung einer SML-Spezifikation und die Simulation des zugehörigen Zustandsgraphen demonstriert. Dabei soll gezeigt werden, wie ScenarioTools verwendet wird, wie neu auftretende Anforderungen in bestehende Projekte implementiert werden können und wie das zu entwickelnde Plugin ScenarioTools verbessern kann.

3.1.1 Demonstration

Die initiale Spezifikation an die Kaffeemaschine beinhaltet zwei Anforderungen. Erstens, dass diese ein- und ausgeschaltet werden kann und zweitens, dass sie Kaffee ausgeben kann. Dazu legt der Ingenieur im ersten Schritt ein neues SML-Project in der ScenarioTools Runtime an. Danach erstellt er das eCore- Modell anhand der Vorlage des Klassenmodells aus Abbildung 2.2. Nachdem er das eCore-Modell erstellt hat, deklariert er dieses als *domain*. Im Anschluss definiert er den Controller als *controllable* und die Komponenten Controller, PowerButton, CoffeeButton und Coffeemachine als *static* analog zu dem Beispiel aus dem zweiten Kapitel. Anschließend setzt er die Rolebindings der Komponenten, wie im zweiten Kapitel beschrieben, in der .runcofing, da alle Komponenten *static* sind.

```
guarantee scenario turnOn{
    powerButton -> controller.powerButtonPressed()
    interrupt[controller.powerOn]
    strict urgent controller -> coffeeMachine.turnOn()
    strict urgent controller -> controller.setPowerOn(true)
}

guarantee scenario turnOff{
    powerButton -> controller.powerButtonPressed()
    interrupt[!controller.powerOn]
    strict urgent controller -> coffeeMachine.turnOff()
    strict urgent controller -> controller.setPowerOn(false)
}

guarantee scenario pourCoffee {
     coffeeButton -> controller.coffeeButtonPressed()
     strict urgent controller -> coffeeMachine.pourCoffee()
}
```

Abbildung 3.1: Die Formulierung der ersten Anforderung in der SML

In Abbildung 3.1 sind die ersten drei Scenarios der SML-Spezifikation der Kaffeemaschine abgebildet. Alle drei Scenarios sind vom Typ *guarantee*, wodurch angezeigt wird, dass es sich um Systemnachrichten handelt.

Das 'turnOn-Scenario' beginnt damit, dass der Power-Knopf bei Betätigung den Controller informiert, dass er gedrückt wurde. War die Maschine schon eingeschaltet, wird durch die *Interrupt*-Bedingung die Ausführung folgender Nachrichten unterbrochen und das 'turnOff'- Scenario ausgeführt. Befand sich die Maschine aber in einem ausgeschalteten Zustand, wird sie nun durch die Nachricht *coffeeMachine.turnOn()* eingeschaltet. Der zugehörige boolean-Wert *powerOn* beschreibt den Zustand der Kaffeemaschine und wechselt seinen Zustand auf true, sobald die Maschine eingeschaltet wird und auf false, wenn sie ausgeschaltet wird. Das 'turnOff'- Scenario enthält das Verhalten der Kaffeemaschine bei Drücken des Power-Buttons, wenn sie eingeschaltet ist. Auch in diesem Scenario wird die Ausführung folgender Nachrichten durch die *Interrupt*- Bedingung unterbrochen und das 'turnOn'-Scenario ausgeführt, wenn die Kaffeemaschine ausgeschaltet war. War sie aber eingeschaltet, wird äquivalent wie im 'turnOn'- Scenario, die Kaffee-

maschine durch die Nachricht *turnOff()* ausgeschaltet und der boolean-Wert *powerOn* auf false gesetzt. Das dritte Scenario enthält das Verhalten der Kaffeemaschine bei Betä- tigung des Kaffeeknopfes. Wird der Kaffeeknopf des Automaten gedrückt, informiert der Controller den Kaffeeautomaten, welcher daraufhin über die Nachricht *coffeeMachine.pourCoffee()* Kaffee ausschenkt. Nachdem der Ingenieur diese drei Scenarios formuliert hat, überprüft er die Spezifikation durch die Play-Out-Simulation in Abbildung 3.2 auf Inkonsistenzen. Die Simulation enthält keine Inkonsistenzen, wodurch gezeigt wird, dass die Spezifikation realisierbar ist.

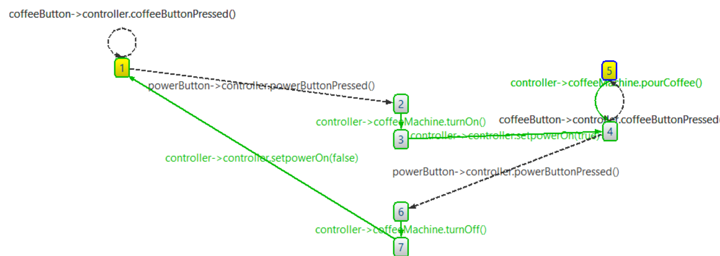

Abbildung 3.2: Simulation der ersten drei Scenarios

Da diese Demonstration dazu dienen soll, das Auftreten und Beheben von Inkonsistenzen zu erklären, wird die Spezifikation der Maschine im folgenden Schritt erweitert. Dazu erhebt der Projektleiter die Anforderung, dass die Kaffeemaschine eingeschaltet sein muss, um Kaffee auszugeben, woraufhin der Ingenieur die SML- Spezifikation erweitert.

```
/*Dont't pour Coffee when the Machine is not turned on */

guarantee scenario OnlyPourWhenPowerOn{
    controller -> coffeeMachine.pourCoffee()
    violation[!controller.powerOn]
    }
}
```

Abbildung 3.3: Die erste Bedingung an den Kaffeeautomaten

In Abbilung 3.3 wird in der SML-Spezifikation durch das *guarantee*-Scenario 'OnlyPourWhenPowerOn' gewährleistet, dass die Maschine nur dann Kaffee ausschenken kann, wenn Sie eingeschaltet ist. Möchte die Kaffeemaschine Kaffee ausschenken, obwohl sie ausgeschaltet ist, wird eine Inkonsistenz generiert.

Um die Umsetzung der Anforderung zu überprüfen, simuliert der Ingenieur die veränderte SML-Spezifikation durch eine Play- Out- Simulation.

Wie in Abbildung 3.4 zu sehen ist, wird im zweiten Zustand eine Inkonsistenz generiert. Anhand der Simulation wird schnell klar, dass das

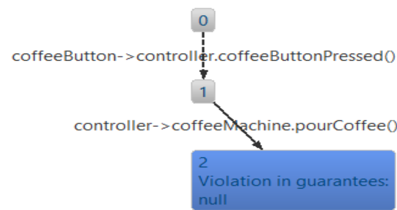

Abbildung 3.4: Die erste Inkonsistenz

Ereignis *coffeeMachine.pourCoffee()* die Inkonsistenz erzeugt hat, indem die Kaffeemaschine Kaffee ausschenken wollte, ohne vorher eingeschaltet worden zu sein. Um die Inkonsistenz zu beheben, ändert der Ingenieur die SML-Spezifikation.

```
/* Fixed Inconsistency. Added the condition not to pour Coffee when no Power */
guarantee scenario pourCoffee {
    coffeeButton -> controller.coffeeButtonPressed()
    interrupt[!controller.powerOn]
    strict urgent controller -> coffeeMachine.pourCoffee()
}
```

Abbildung 3.5: Beheben der ersten Inkonsistenz

In Abbildung 3.5 ist die modifizierte SML-Spezifikation abgebildet. Das 'pourCoffee'- Scenario wurde um eine *Interrupt*- Bedingung erweitert, die die Ausgabe des Kaffees unterbricht, wenn die Kaffeemaschine ausgeschaltet ist.

Um das Verständnis von Inkonsistenzen noch weiter zu fördern, wird die Spezifikation um eine weitere Bedingung erweitert. Dazu erhebt der Projektleiter im nächsten Meeting die Anforderung, dass der Kaffee nur dann ausgeschenkt werden darf, wenn eine Tasse bereit steht.

```
guarantee scenario cupDetection{
    cupsensor -> controller.cupDetected
    interrupt[controller.cupDetected]
    strict urgent controller -> controller.setCupDetected(true)
}
```

Abbildung 3.6: Sensorik der Tassendetektion

Der Ingenieur erkennt, dass ein neues Element benötigt wird, um die Anforderung umzusetzen. Dazu erweitert er das eCore- Modell um den 'CupSensor' und erstellt, wie in Abbildung 3.6 abgebildet, ein Scenario, dass sein Verhalten beschreibt.

Des weiteren erweitert er die *Violation*- Abfrage, um die Bedingung, dass eine Tasse von der Sensorik der Kaffeemaschine erkannt werden muss,

```
/*Dont't pour Coffee when the Machine is not turned on and no Cup is sensored*/

guarantee scenario OnlyPourPowerOnAndWhenCupBeneathValve{
    controller -> coffeeMachine.pourCoffee()
    violation[!controller.cupDetected || !controller.powerOn]
    }
```

Abbildung 3.7: Erweitern der Violation-Abfrage

bevor sie Kaffee ausgibt. Anschließend überprüft er die Erweiterung in einer Simulation.

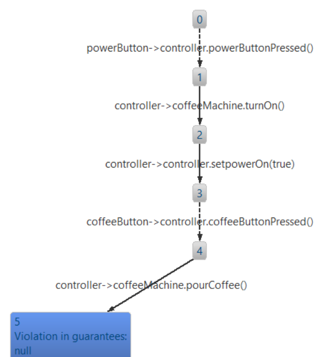

Abbildung 3.8: Die zweite Inkonsistenz

Wieder wird durch das Event *coffeeMachine.pourCoffee()* eine Inkonsistenz erzeugt. Der Ingenieur erinnert sich daran, dass die erste Inkonsistenz auch durch dieselbe Nachricht erzeugt wurde, und er die *Interrupt*- Bedingung erweitert hat, um diese zu beheben. Aufgrund dessen nimmt er an, dass die Inkonsistenz wie bei der ersten Bedingung durch eine Erweiterungen der *Interrupt*-Bedingung behoben werden kann.

Der Ingenieur lag richtig. Durch die Erweiterung der *Interrupt*-Bedingung ist die Spezifikation wieder realisierbar.

In diesem Beispiel bestand die Erweiterung nur aus wenigen Komponenten. Daher war es verhältnismäßig einfach, wiederkehrende Inkonsistenzen zu erkennen, einzuordnen und die SML-Spezifikation zu korrigieren. In der Realität bestehen Spezifikationen von Systemen meist aus mehr als drei Szenarien, wodurch ihr Komplexitätsgrad, vor allem bei Modifizierung des Systemverhaltens, weitaus höher ist. Folglich kann der Fall, dass Änderungen oder Erweiterungen der Spezifikation die Quelle der Inkonsistenzen verlagern oder zu neuen Inkonsistenzen führen, mit einer größeren Wahrscheinlichkeit eintreten. In diesen Fällen kann trotz eines Gespürs für die Charakterisierungen von Inkonsistenzen leicht der Überblick darüber verloren werden, ob es sich um neue oder um wiederkehrende Arten von Inkonsistenzen handelt.

```
/*Cup Detection */
guarantee scenario SetCup{
    coffeeButton -> controller.coffeeButtonPressed()
    interrupt[!controller.cupDetected]
    strict urgent controller -> controller.setCupDetected(true)
}

/* Fixed Inconsistency. Added the condition not to pour Coffee when no Cup is detected */
guarantee scenario pourCoffee {
     coffeeButton -> controller.coffeeButtonPressed()
     interrupt[!controller.powerOn || !controller.cupDetected]
     strict urgent controller -> coffeeMachine.pourCoffee()
}
```

Abbildung 3.9: Beheben der zweiten Inkonsistenz

3.1.2 Anforderungen

Durch die Demonstration im vorigen Abschnitt wird schnell ersichtlich, dass das Plugin Inkonsistenzen anzeigen und charakterisieren können muss, um beim Auftreten neuer Inkonsistenzen Vergleichswerte zu haben und dadurch eine schnelle Zuordnung der entstandenen Inkonsistenzen in neu oder wiederkehrend vornehmen zu können. Weiterhin erstreckt sich in Softwareunternehmen die Entwicklung von Anwendungen über einen längeren Zeitraum. Daher muss der Arbeitsfortschritt abgespeichert und geladen werden können. Außerdem muss bei einer großen Anzahl von Inkonsistenzen die Möglichkeit bestehen, die diese einem Parameter nach zu gruppieren, um wiederkehrende Inkonsistenzen zu identifizieren. Konkret haben sich aus den Überlegungen die folgenden Anforderungen ergeben:

a) **Inkonsistenzen müssen anhand von fest definierten Kriterien charakterisiert werden können**

   Damit die Charakterisierung der Inkonsistenzen effektiv durchgeführt werden kann, ist es erforderlich, gesonderte Auswahlkriterien zu treffen, nach denen die Inkonsistenzen identifiziert werden sollen. Damit das Finden der Inkonsistenzen im Zustandsgraph erleichtert wird, ist es notwendig, den Pfad vom Ursprung zur Inkonsistenz zu notieren. Um die kognitive Arbeitslast zu reduzieren und einen Sortierparameter zu erhalten, wird die Nachricht, die die Inkonsistenz auslöst - wie z.B. *coffeeMachine.pourCoffee()* aus dem Beispiel des Ingenieurs- , als weiteres Charakterisierungskriterium verwendet. Desweiteren kann durch die Betrachtung der Kombination aus Pfad und Nachricht ein besseres Verständnis dafür entwickelt werden, wie die Spezifikation durch die Simulation umgesetzt wurde und wie sie verletzt wurde. Der Arbeitsaufwand kann noch weiter gesenkt werden, indem bei *safety*-Violations zusätzlich noch aufgezeigt wird, welche alternativen

Events im vorletzten Zustand eines Violation-behafteten Pfades möglich waren, um ggf. die Formulierung der inkonsistenten Nachricht an diese anzupassen. Bei *Llveness*-Violations ist es sinnvoll, den Zyklus anzugeben, der die Inkonsistenz verursacht. Weiterhin ist es hilfreich, den Typ der Inkonsistenz zu notieren. Dadurch kann ohne eine weitere Information die Eigenschaften der Inkonsistenz identifiziert werden.

b) **Inkonsistenzen müssen übersichtlich angezeigt werden können**

Eine effiziente Form zur Darstellung von Daten ist die Verwendung von Tabellen. Tabellen haben den Vorteil, dass sie durch ihre Struktur eine klare Übersicht des Inhalts geben können. Außerdem muss die Tabelle in ScenarioTools aufgerufen werden können, damit der Ingenieur während der Benutzung des Tools die Tabelle verwenden kann.

c) **Inkonsistenzen müssen anhand von Parametern gruppiert werden können**

Um bei vielen Inkonsistenzen den Überblick zu behalten, soll der Inhalt der Tabelle nach der Spalte der Inkonsistenz- verursachenden Nachricht sortiert werden können, damit eine Gruppierung der Inkonsistenzen durchgeführt werden kann.

d) **Inkonsistenzen müssen abgespeichert und geladen werden können**

Die Arbeit an Projekten erstreckt sich in der Regel über einen längeren Zeitraum. Daher ist es notwendig, dass der Arbeitsfortschritt abgespeichert und zu einem späteren Zeitpunkt wieder geladen werden kann.

e) **Kommentare müssen in die Anzeige eingefügt werden können**

Der Ingenieur soll in der Lage sein, zu jeder Inkonsistenz einen Kommentar zu erfassen, diesen mit der Inkonsistenz zu speichern und zu einem späteren Zeitpunkt zu laden.

3.2 Entwurf erstellen

3.2.1 Anforderungen zusammenfassen und klassifizieren

Anforderungen unterscheiden sich in der Art ihrer Forderung an das System. Nach der Definition aus dem Buch 'Requirements Engineering' von Hull, Jackson und Dick aus dem Jahre(2011)[3] wird zwischen funktionalen-, nicht-funktionalen- und Designanforderungen unterschieden. Angewandt auf die oben formulierten Anforderungen an das Plugin, sind alle fünf Anforderungen funktionaler Natur, weil sie die Funktionalität des Plugins beschreiben.

3.2.2 Prototyp des GUI

Eine Form, Anforderungen zusammenzufassen, ist die Verwendung von Prototypen. Bei der Verwendung von Prototypen wird auf Details verzichtet, um den Fokus auf die Funktionen zu legen. Sie repräsentieren Meilensteine, indem sie die Entwicklung bis zu einem bestimmten Punkt darstellen können[3]. Um die oben erhobenen Anforderungen zusammenfassend darzustellen, wird im folgenden ein Prototyp der Tabelle aufgeführt, der als Grundlage der Implementierung verwendet wird.

Abbildung 3.10: Erfüllen der Anforderungen durch einen Prototyp

In Abbildung 3.10 ist der Prototyp der Tabelle zu sehen. Die Tabelle enthält fünf Spalten. In der ersten Spalte ist der Pfad der Inkonsistenz von Startzustand bis zum inkonsistenzen Zustand vermerkt. Die zweite Spalte enthält die Indizes der kritischen Transition. In der dritten Spalte ist das Event aufgelistet, welches anstelle des Inkonsistenz- verursachenden Events eintreten könnte. Dabei soll für jedes mögliche alternative Event ein Eintrag erstellt werden, sodass auch gezeigt wird, welche Alternativen es gibt. Die vierte Spalte zeigt den Typ der Inkonsistenz und kann die Werte ’Safety’ oder ’Liveness’ annehmen. In der letzten Spalte können Kommentare eingefügt werden. Unter der Tabelle sind zwei Buttons zu finden. Bei Betätigung des ’Safe Data’- Buttons werden die Tabelleninhalte in einer Datei abgespeichert und der ’Load Data’- Button lädt die Inhalte der abgespeicherten Inhalte in die Tabelle.

3.3 Verwendete Verfahren

Neben der Verwendung von Java, und den Entwicklungstools Eclipse und ScenarioTools, werden bei der Implementierung und der Entwicklung des Plugins allgemeingültige Muster benutzt, die in diesem Teilkapitel vorgestellt und erklärt werden. Dazu gliedert sich dieses Unterkapitel in drei Teile. Im ersten Teil wird das MVC-Pattern vorgestellt. Das MVC-Pattern ist ein Design-Pattern, welches eine Anwendung in die zwei Komponenten Model, View und Controller unterteilt, um somit eine flexiblere und wiederverwertbare Konstruktion der Anwendung zu ermöglichen. Im zweiten Teil wird die Tiefensuche erklärt. Die Tiefensuche ist ein Algorithmus, welcher verwendet wird, um Baumstrukturen zu durchsuchen.

3.3.1 MVC-Pattern

Ein Verfahren, das von erfahrenen Designern eingesetzt wird, ist die Wiederverwendung von funktionierenden Abläufen. In vielen Objektorientierten Systemen, Problemstellungen und Algorithmen kann wiederkehrendes Verhalten beobachten werden. Dadurch wurden die sogenannten Design-Patterns entwickelt, die eine Schablone für die Lösung bestimmter Probleme bereitstellen. Eines dieser Pattern ist das MVC-Pattern. Das MVC-Pattern oder auch Model-View-Controller-Pattern besteht aus den drei Komponenten Model, View und Controller[5]. Das Model enthält die Datenstrukturen, welche in der Anwendung verwendet werden. Im allgemeinen werden im Model *getter()* und *setter()* Methoden und eine *Konstruktionsfunktion* für die Erstellung von Model-Objekten formuliert. Der View beinhaltet die Bereitstellung einer Schnittstelle zwischen Mensch und Maschine in Form eines Graphical User Interfaces(GUI). Im Controller wird die Verwaltung der Daten durchgeführt. Bei einer Veränderung der Daten im View, informiert der Controller das Model, wodurch dieses die Daten ändert und über den Controller zurück an den View sendet, wo diese dann dargestellt werden. Durch eine niedrige Kopplung zwischen den Klassen, die durch die Dezentralisierung der Funktionalität erreicht wird, und einer hohen Kohäsion innerhalb der Klassen, wird eine niedrige Abhängigkeit und ein hoher Abdeckungsgrad der Funktionalität erreicht. Weitere Informationen über Design-Patterns können in dem Buch 'Design Patterns'[5] gefunden werden.

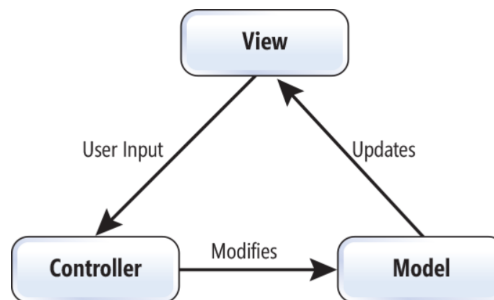

Abbildung 3.11: MVC-Pattern

In Abbildung 3.11 ist die Kommunikation zwischen den drei Komponenten Model, View und Controller zu sehen.

3.3.2 Tiefensuche

Die Tiefensuche ist ein Algorithmus, der dafür verwendet wird, Strukturen wie Bäume zu durchsuchen. Dabei startet der Algorithmus im Startzustand und verfolgt einen Pfad bis zu seinem Endzustand. Dabei markiert er alle

Knoten, die er schon besucht hat. Hat er das Ende eines Pfades erreicht, geht der Algorithmus bis zum ersten Zustand zurück, der eine Abzweigung hat. Ausgehend von diesem Knoten durchsucht er alle Nachfolger des Knoten in derselben Art und Weise[19]. Der Nachteil der Tiefensuche liegt darin, dass sie nicht *vollständig* arbeitet. Das bedeutet, dass sie bei Auftreten von Zyklen oder bei unendlich großen Graphen nicht terminiert, obwohl es ein Ergebnis gibt.

3.3.3 Beschränkte Tiefensuche

Die beschränkte Tiefensuche erweitert die normale Tiefensuche um die Begrenzung der Suchtiefe. Dadurch kann vermieden werden, unendliche Baumtiefen zu durchsuchen. Weiterhin kann das Kriterium der *Vollständikeit* durch das Angeben einer Tiefe, in der sich das Ziel befindet, gewährleistet werden.

Algorithm 1 Beschränkte Tiefensuche

```
1: BeschränkteTiefensuche(Node N, Goal D, Depth D)
2: if (N == G) then
3:    Return N;
4: end if
5: Stack := Expand (N)
6: while (Stack is not empty) do
7:    Node N' := Pop(Stack)
8:    if (N'.depth() < D) then
9:       BeschränkteTiefensuche(N', G, D)
10:   end if
11: end while
```

Wie im Pseudocode abgebildet, arbeitet die beschränkten Tiefensuche rekursiv. Im ersten Schritt wird überprüft, ob das Ziel erreicht wurde und der Zustand zurückgegeben. Ist der aktuelle Zustand N nicht das Ziel, so werden alle Nachfolger von N auf den Stack gelegt und für jeden Folgezustand, dessen Tiefe kleiner als die Zieltiefe ist, die Funktion erneut aufgerufen. Auf diese Art und Weise wird ein Baum bis zu einer vorbestimmten Tiefe durchsucht.

Kapitel 4

Implementierung

Eine Implementierung setzt unter Beachtung von Randbedingungen und Spezifikationen durch das Konkretisieren von Abstraktionen Entwürfe in reale Konstrukte um. Sie repräsentiert die tiefste Ebene in der Entwicklung und ist gleichzeitig auch die dritte Phase des Wasserfallmodells.[20]

Während der Implementierung werden die im Konzept aufgestellten Anforderungen und Entwürfe aus Kapitel 3 in Java umgesetzt, das Plugin entwickelt und in die bestehende ScenarioTools-Umgebung integriert. Dazu gliedert sich die Implementierung in zwei Teile. Der erste Teil handelt von der Konstruktion des Plugins zur Erfüllung der Anforderungen, sodass dieses unabhängig von der Integration in ScenarioTools eine Benutzerschnittstelle in Form einer Tabelle bereitstellt und mit manuell erstellten Daten funktioniert[5].

Im zweiten Teil wird das Plugin in die ScenarioTools Umgebung integriert, indem die Schnittstellen an den bestehenden ScenarioTools-Code angepasst werden und die Anzeige des Plugins über das ScenarioTools-Framework umgesetzt wird. Abschließend wird das Zusammenspiel der Klassen in Form eines UML-Klassendiagramms visualisiert.

4.1 Implementierung part One: Implementieren der Anforderungen

Der erste Schritt der Implementierung bestand aus dem Anlegen eines neuen Java- Plugins. Anschließend wurden die benötigten Klassen erstellt. Da das Plugin im MVC- Pattern entstehen sollte, wurden zu Beginn die Klassen Inconsistency, Table und Controller erstellt, die repräsentativ für Model, View und Controller stehen.

4.1.1 Anforderung a): Charakterisieren der Inkonsistenzen

Die Charakterisierung der Inkonsistenzen findet sich im Model des Plugins wieder. Wie in der Anforderung beschrieben, sollen Inkonsistenzen fünf Parameter aufweisen. Dazu wird für jeden der Parameter Pfad, kritisches Ereignis, alternative Ereignisse, Typ der Inkonsistenz und Kommentarfeld ein neuer String in der Klasse Model angelegt. Im Unterschied zu dem Prototypen aus Kapitel drei wurde das Model der Inkonsistenz um ein sechstes Element erweitert. Dieses Element ist ein boolean und wird in Anforderung b) erläutert.

Durch die *public* Deklaration und einem Konstruktor können Inkonsistenzen- instanzen in anderen Klassen erstellt werden. Des weiteren können die einzelnen Elemente einer Inkonsistenz durch *getter()*- und *setter()*-Methoden gesetzt und abgerufen werden.

4.1.2 Anforderung b): Anzeigen der Inkonsistenzen

Wie in Kapitel 3 beschrieben und im Prototyp aufgeführt, erfolgt die Anzeige der Inkonsistenzen in Form einer Tabelle. Ein Weg zur Erstellung von Tabellen als Graphical User Interfaces(GUI) ist, wie von Moessenboeck erklärt[17], die Verwendung der Java Abstract Windowing Toolkit(AWT) und Java Swing Bibliotheken. Durch die Hinzunahme vom JTable und dem DefaultTableModel ist es möglich, eigene Tabellenmodelle zu erstellen, die eingebaute Funktionen wie z.B. die automatische Generierung von Kopfzeilen, das Einfügen von JComponents oder die Möglichkeit, Einträge editieren zu können, bereitstellen[18].

Die Erstellung des GUI beginnt mit der Initialisierung eines Mainframes. Dem Mainframe wird dann ein Mainpanel übergeben, das im Border- Layout angelegt wird. Diese wiederum enthält die Tabelle im zentralen Bereich und JButtons zum Manipulieren der Tabelleninhalte im südlichen Bereich enthält. Die Tabelle wird mit JTable erstellt, die bei Konstruktion als Übergabeparameter den Datensatz und die Spaltennamen übergeben bekommt. Der Datensatz sind die Inkonsistenzen, die in Form einer Array- Liste abgespeichert werden. Die Spaltennamen definieren die Spalten der Tabelle. Des weiteren fungieren die oben erwähnten JButtons als Kontrollelement der Tabelle. Sie sind mit einem ActionListener versehen, der bei Betätigung die Funktionen des Sortierens, Löschens, Speicherns und Ladens ausführen kann. Der Listener befindet sich in der Controller Klasse.

In Abbildung 4.1 ist das GUI des Plugins zu sehen. Im Unterschied zu dem Prototyp aus Kapitel 3 wurden eine Spalte, ein Feld für Text und zwei Buttons hinzugefügt. Die neue Spalte enthält, wie in der Implementierung der ersten Anforderung angedeutet, eine JCheckBox. Durch die Funktion der JTable, booleans in JCheckBoxes umzuwandeln zu können, wird der sechste Eintrag im Model durch einen *boolean* definiert. Das hinzugefügte

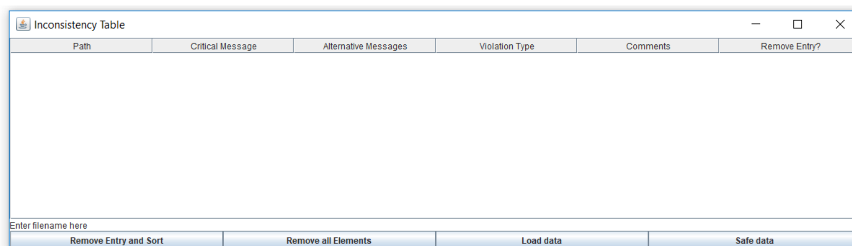

Abbildung 4.1: GUI des Plugins

Textfeld ermöglicht das Speichern und Laden von verschiedenen Dateien. Die beiden neuen Buttons erweitern die Funktionalität des GUI. Die Betätigung des 'Remove Entry and Sort'- Buttons, resultiert in dem Entfernen der Einträge, deren boolean *Remove Entry* auf *true* gesetzt wurden, und dem Sortieren der Tabelle anhand des kritischen Ereignis. Beide Funktionen werden über Funktionen realisiert, die durch den zugehörigen ActionListener aufgerufen werden. Die Sortierung wird im Unterkapitel Anforderung c) näher beschrieben. Der 'Remove All Elements'- Button entfernt alle Elemente aus der Tabelle. Auch diese Funktion wird durch den Aufruf über den ActionListener ausgeführt. Die beiden übernommenen Buttons enthalten in Kombination mit der JTextArea die Lade- und Speicherungsfunktion und werden in der Implementierung von Anforderung d) beschrieben.

4.1.3 Anforderung c): Sortieren der Inkonsistenzen

Da die Inkonsistenzen nach dem Event, welches die Inkonsistenz auslöst, gruppiert werden sollen, wird die Spalte 'Critical Event' als erstes, 'Path' als zweites und 'Alternative Messages' als drittes Sortierkriterium verwendet. Die Methode *.sort(ArrayList<Inconsistency>)* verwendet die Methode *.sort(Collection, new Comparator<Object>)* der Collections Bibliothek von Java. Diese wiederum enthält die *compare(Object,Object)*-Funktion, die zwei Objekte übergeben bekommt. Das Sortieren wird durch die *Object.String.compareTo(Object.String)*- Methode, die in der *compare(Object,Object)*- Methode aufgerufen wird und die Strings der beiden Objekte vergleicht[18], durchgeführt. Um die Tabelle den drei Kriterien nach zu sortieren, wird die *Object.String.compareTo(Object.String)* bis zu dreimal immer dann aufgerufen, wenn die Vergleichswerte gleich sind. Wie oben erwähnt wird durch das Betätigen des 'Remove Entry and Sort'- Button die *.sort(ArrayList<Model>)* Methode aufgerufen, die das Sortieren der Array Liste ausführt.

4.1.4 Anforderung d): Abspeichern und Laden der Inkonsistenzen

Die Speicherung in externe Dateien erfolgt in Java durch die Verwendung des *FileWriters* aus der Utility Bibliothek. Dieser bekommt als Parameter einen Dateinamen oder einen Pfad übergeben, der angibt, wie die Datei heißen soll, welchen Typs sie ist und wo sie gespeichert werden soll. Mit der Funktion *FileWriter.append(String)* kann die Datei beschrieben werden. Durch eine Iteration über die Tabelleninhalte und zeilenweiser Speicherung des Inhalts in Strings, welche durch die *toString()-Methode* der Model Klasse erstellt und um einen Zeilenumbruch erweitert werden, wird die Datei mit den Inhalten den Tabelle beschrieben. Durch die Betätigung des Safe-Buttons wird der implementierte ActionListener der Klasse aufgerufen, der die Methode *storeData(ArrayList<Model>)* aufruft, die den Tabelleninhalt der Comma seperated Value - Datei hinzufügt.

Das Laden der externen Datei erfolgt durch den *FileReader*, der auch aus der Utility Bibliothek von java kommt. Der *FileReader* bekommt als Übergabeparameter einen Dateinamen, wenn sich die Datei im aktuellen Verzeichnis befindet, oder den direkten Pfad zu einer Datei übergeben. Dann wird der *FileReader* in einen *BufferedReader* übersetzt, weil dieser in der Lage ist, Dateien zeilenweise einzulesen. Durch eine while Schleife, die mit der Funktion *.readLine()* solange Zeilen einliest und diese durch die *String.split(String)*-Funktion in seine Komponenten unterteilen kann, bis die Datei komplett durchgelesen ist, können die Inkonsistenzen in Form von Model-Objekten konstruiert werden. Die implementierte Funktion des Load-Buttons ruft den ActionListener auf, der wiederum die Methode *getData(String)* des TableModels aufruft, die den Dateinamen übergeben bekommt, und die Tabelle mit den Elementen der Datei füllt.

Der in der Beschreibung von Lade- und Speicherfunktion erwähnte Übergabeparameter des Dateinamen wird durch das Textfeld realisiert. Bei Betätigung des Lade- oder Speicherknopfes wird der String des Textfeldes als Filename des FileReader und FileWriter verwendet. Das Verzeichnis ist dabei im Code definiert und kann bei Bedarf geändert werden. Dadurch kann gewährleistet werden, dass mehrere Dateien parallel existieren können, um z.B. verschiedene Projekte zu bearbeiten. Wenn bei Betätigung des Load-Buttons ein Dateiname eingegeben wird, der nicht existiert, wird ein JDialog erstellt, der dem Benutzer mitteilt, dass es keine Datei unter dem angegeben Namen gibt. Ebenso wird bei der Betätigung des Safe- Buttons eine Warnung in Form eines JDialogs generiert, der nachfragt, ob die Datei überschrieben werden soll.

4.1.5 Anforderung e): Editiermöglichkeit der Anzeige

Die Editiermöglichkeit von Tabelleninhalten kann durch die eingebaute Funktion *isCellEditable(int, int)* des Tabellenmodels der JTable umgesetzt werden. Durch die Auswahl eines Tabellenelements kann dieses, nachdem es durch *isCellEditable(int,int)* zur Modifizierung freigegeben ist, beschrieben werden, um Notizen, Kommentare oder Hinweise für die spätere Bearbeitung einzufügen. Dazu wurde die Spalte 'Comments' eingefügt, die die einzige Spalte ist, deren Zellen editierbar sind.

4.1.6 Anmerkungen

Die Funktionalität im MVC-Verbund wird durch den Controller realisiert. In der Implementierung des Plugins wurde der Controller durch den implementierten ActionListener des Views umgesetzt. Dieser kontrolliert das Ausführen der Methoden durch die Möglichkeit auf Buttonbetätigungen zu reagieren. Die Methoden sind in der Klasse Methods.java ausgelagert, um eine übersichtliche Struktur zu generieren.

4.2 Implementierung part Two: Kopplung mit ScenarioTools

Der zweite Teil der Implementierung hatte das Ziel, das funktionierende Plugin mit dem dem Tool zu koppeln. Dazu wurde nacheinander erst die Anzeige des GUI durch die Funktionalität des ScenarioTools Frameworks und dann das Finden der Inkonsistenzen in einem SMLRuntimeGraph in Form einer Methode umgesetzt.

4.2.1 Kopplung des Views

Die Klasse *StateGraphViewPart.java.* enthält die Anzeige der Buttons des State-Graph-Views und musste um einen Button zur Anzeige der Inkonsistenztabelle über das Plugin erweitert werden. Dazu musste als erstes eine neue Action createInconsistencyAction angelegt werden. Im nächsten Schritt wurde in der Methode makeActions() der *StateGraphViewPart.java.* die Action createInconsistencyTableAction erstellt und durch den Aufruf der createInconsistencyTable()-Methode, welche als Rückgabewert eine Action hat, initialisiert. Weiterhin wurde das ActionContributionItem inconsistencyTable durch new ActionContributionItem(createInconsistencyTableAction), also durch die Action, die durch createInconsistencyTable()- Methode übergeben wird, initialisiert. Die Aktionen sind hierbei die Elemente, die als Funktion während der Anzeige des Stategraphes ausgewählt werden können. Die Konstruktorfunktion der Action enthält eine run()-Methode,

welche bei Auswahl des Plugins gestartet wird. In dieser run()- Methode befindet sich die Initialisierung eines View Objekts. Weiterhin wird in der run()-Methode die Tabelle durch *.show()* angezeigt, und durch *.getInconsistencies(SMLRuntimeStateGraph)* werden die Inkonsistenzen des übergebenen SMLRuntimeStateGraphen ermittelt und der Tabelle hinzugefügt. Des weiteren enthält die createInconsistencyTable-Methode die Methoden *.setImageDescriptor(Activator.getImageDescriptor(imgage))*, welches dem Plugin im StateGraphView ein Bild beifügt, und *.setEnabled(true)*, wodurch die Action aktiviert wird.

Damit die Auswahl der Methode getroffen werden konnte, musste die Action im letzten Schritt der Liste von ContributionItems in der getPullDownActions Methode, welche alle Funktionen des Aufklappmenüs enthält, hinzugefügt werden.

4.2.2 Kopplung der Funktionalität

Die Kommunikation läuft über die Schnittstelle der *getInconsistencies(SMLRuntimeStateGraph)*- Methode. Diese bekommt als Übergabeparameter den SMLRuntimeGraph des aktuellen Zustandsgraphen übergeben. Um die Inkonsistenzen in der Tabelle anzeigen zu können, müssen die nötigen Informationen aus dem Graphen extrahiert werden. Die *getInconsistencies(SMLRuntimeStateGraph)*-Methode repräsentiert die algorithmische Programmierung und ist gleichzeitig auch die Methode, die den Graphen auf Inkonsistenzen durchsucht. Der Methode wird bei Aufruf der aktuelle SMLRuntimeStateGraph übergeben. Die Wurzel des Zustandsgraphen wird in Form einer ArrayListe<SMLRuntimeState> angelegt und dem Stack als erster Pfad hinzugefügt. Ausgehend von diesem Knoten werden alle weiteren Pfade durch die Tiefensuche konstruiert. Diese wiederum wird durch eine while- Schleife initialisiert, die in jedem Durchlauf den obersten Pfad des Stacks als aktuellen Pfad definiert und solange läuft, bis keine Pfade mehr auf dem Stack liegen.

Innerhalb der while- Schleife gilt es diesen Pfad auf Inkonsistenzen zu überprüfen. Die Überprüfung auf Inkonsistenzen erfolgt in zwei Schritten. Im ersten Teil des Algorithmus wird der Pfad auf *safety*- Violations überprüft. Durch die Funktion *SMLRuntimeState.isSafetyViolationinGuarantees()* und *SMLRuntimeState.isSafetyViolationinAssumptions()* kann überprüft werden, ob ein Zustand eine *safety*- Inkonsistenz aufweist. Durch die Iteration über die Zustände des Pfades kann dadurch gewährleistet werden, dass alle *safety*- Violations gefunden werden. Tritt der Fall ein, dass ein Zustand eine *safety*- Violation aufweist, wird ein Objekt des Typs Model erstellt. Der Pfad wird durch die Iteration über alle Zustände und der Bestimmung des Indexes des jeweiligen Zustandes mithilfe der Funktion *SML.RuntimeStateUtil.getPassedIndex(SMLRuntimeZustand)* ermittelt. Die kritische Transition wird mithilfe der *fromTo(SMLRuntimeState, SMLRun-*

timeState)- Funktion ermittelt, die den Übergang der beiden übergebenen Zustände ermittelt und als Transition- Objekt zurückgibt. Aus diesem kann dann durch die Funktion *.getLabel()* die Nachricht der Transition ermittelt werden. Die alternativen Messages, die anstelle der kritischen hätten ausgewählt werden können, werden durch die Iteration über alle ausgehenden Transitionen des vorletzten Zustandes generiert. Dabei wird durch die Funktion *.getOutgoingTransition()* eine Liste des Typs Transition bereitgestellt, die alle ausgehenden Transitionen enthält. Jede dieser Transitionen wird dann in eine neue Inkonsistenz umgesetzt. Im Feld des Violation Typs wird 'Safety' eingetragen und der boolean *Remove Entry?* auf false gesetzt, da der Eintrag noch nicht gelöscht werden soll. Das Kommentarfeld wird mit dem Text 'Place for Comments' initialisiert, wodurch dem Nutzer signalisiert werden soll, dass er diese Spalte editieren kann.

Nachdem der Pfad auf *safety*- Violations überprüft wurde, wird er auf *liveness*- Volations untersucht. Wie in Kapitel 2 beschrieben, impliziert eine *liveness*- Violation das Auftreten eines Zyklus innerhalb des Pfades. Da der Pfad durch den Zyklus beschrieben wird, muss das letzte Element des Pfades innerhalb des Pfades mindestens ein zweites Mal vorkommen, damit ein Zyklus entsteht. Tritt der Fall ein, so wird der Zyklus aus dem aktuellen Pfad extrahiert und als aktueller Pfad definiert. Dann werden für jeden Zustand des Pfads alle *enabled Messages* aller aktiven Scenarios, die mit Modalitäten versehen wurden, in einer Liste gespeichert. Dazu wird die Funktion *getMessage()* verwendet. Diese sucht rekursiv nach allen Nachrichten, die mit Modalitäten versehen sind, speichert diese in einem Set ab und gibt dieses zurück. Die Rekursion ist notwendig, da die ActiveParts, die die Interaktion der Systemkomponenten enthalten, neben dem Typ *ModalMessage* weitere Typen, die im Unterschied zur *ModalMessage* wieder einen ActivePart enthalten können, annehmen können. Im nächsten Schritt wird dann die Schnittmenge der Message- Listen aller Zustände verglichen. Ist die Schnittmenge leer, so ist kein Scenario über die gesamte Zykluslänge aktiv, wodurch eine Liveness-Violation ausgeschlossen werden kann. Ist die Schnittmenge aber nicht leer, so wird ein Model- Objekt erstellt, das neben dem Violation-Typ, dem Kommentarfeld und der JCheckBox nur den Zyklus als Pfad enthält, und der Tabelle hinzugefügt.

Um zu verhindern, dass der Stack nicht geleert wird, ist es wichtig, dass die Folgezustände der Pfade, die Zyklen enthalten, nicht weiter auf den Stack gelegt werden, da diese auch wieder Zyklen enthalten. Dazu wurde nach der Überprüfung eines Zyklus behafteten Pfades ein *continue* eingebaut, welches den Parser vor den Beginn der while-Schleife positioniert. Enthielt der Pfad allerdings keinen Zyklus so werden im letzten Schritt alle Pfade konstruiert, die durch das Hinzufügen eines Folgeknotens konstruiert werden können. Dazu wird über die Folgezustände des letzten Zustands iteriert und für jeden Folgezustand ein neuer Pfad auf den Stack gelegt.

Kapitel 5

Evaluation

Die Bewertung von Projekten erfordert die Analyse aus verschiedenen Blickwinkeln. Bei z.B. einem Tanzwettbewerb wird der Tänzer von verschiedenen Juroren bewertet, die jeweils ein anderes Kriterium wie Technik, Rhythmus oder Mimik betrachten. Ähnlich verhält es sich mit der Betrachtung eines Projekts. Vor allem das Graphical User Interface(GUI), das als Schnittstelle zwischen Mensch und Maschine fungiert, aber auch die Funktionalität und die Bedienbarkeit sind Kriterien, die bei der Bewertung des Plugins zum Tragen kommen.

Um diese Kriterien des Plugins zu bewerten, gliedert sich die Evaluation in zwei Bereiche. Im ersten Teil wird das Interface des Plugins analysiert. Dazu wird zuerst eine heuristische Evaluation mithilfe der zehn Heuristiken Jacob Nielsens durchgeführt[10]. Danach wird das GUI anhand der ISO Norm 9241-110, die Anforderungen an grafische Benutzerschnittstellen bschreibt, analysiert. Die detaillierte Erläuterung beider Verfahren lassen sich in dem Buch 'Methoden zur Evaluation von Software' erklärt[10]. Anschließend wird in einer Umfrage, die sowohl Kriterien von Jacob Nielsens Heuristiken als auch Anforderungen der ISO 9241-110 enthält, das Interface von potentiellen Benutzern bewertet.

Im zweiten Teil wird die Funktionalität des Plugins evaluiert. Dabei wird zu Beginn anhand eines Cognitive Walkthroughs die Funktionalität aus der Sicht eines potentiellen Nutzers analysiert. Zweck des Cognitive Walkthrough ist es, zu erläutern, welche Probleme bei der Benutzung des Plugins auftreten und wie diese Probleme behoben werden können[10]. Im zweiten Abschnitt werden die drei Usability- Elemente des Plugins - Effizienz, Effektivität und Zufriedenstellung -, wie sie in der ISO 9241 in Teil 11 definiert sind, ausgewertet[10] und anhand eine Experimentbeschreibung aufgezeigt, wie das Plugin zu einer Effizienzsteigerung führen könnte. Danach wird eine SOLL-IST Analyse der Funktionalität durchgeführt. Während der SOLL-IST Analyse wird verglichen, ob die anfangs aufgestellten Anforderungen erfüllt wurden.

5.1 Evaluation des GUI

5.1.1 Heuristische Evaluation

Heuristiken sind teilweise fundierte Annahmen über die Funktionalität eines Systems. Sie basieren auf Erfahrungen, die durch das Ausführen einer Tätigkeit gewonnen werden. Die von Jacob Nielsen und Rolf Molich zehn aufgestellten Heuristiken sind allgemein anerkannte Prinzipien bei der Analyse einer Mensch-Maschine-Schnittstelle[10]. Die Vorteile liegen darin, dass die Evaluation durch Personen ausgeführt werden kann, die nicht an der Entwicklung beteiligt waren und in der Regel eine kritischere Position einnehmen, wodurch objektive Ergebnisse möglich sind[10].

- Sichtbarkeit des Systemstatus: Weiß der Benutzer, an welcher Stelle des Systems er sich befindet?

Bei der Betrachtung der Tabelle wird dem Benutzer schnell klar, dass er sich in einem Bereich des Systems befindet, in dem Daten zu Szenarien und Inkonsistenzen dargestellt werden. Durch die intuitive Interpretation der Spaltenbeschriftungen kann gefolgert werden, dass die Daten aus einem Graphen, welcher die Modellierung eines Systems visualisiert, entnommen wurden. Weiterhin kann durch das Eingabefeld und den Lade- und Speicherbutton gefolgert werden, dass die Inhalte der Tabelle geladen und gespeichert werden können.

- Übereinstimmung zwischen dem System und der realen Welt: Werden Analogien zu realen Welt hergestellt?

Durch die Interpretation der Spaltenüberschriften kann der Benutzer einen Bezug zur realen Welt erstellen. Ein Pfad deutet darauf hin, dass es ein Weg zwischen zwei Punkten gibt. Die kritische Nachricht kann eine Stelle auf dem Pfad sein, die das weitere Betreten des Pfades unterbricht, während alternative Pfade dem Benutzer die Möglichkeit geben, die kritische Stelle eines Pfades zu umgehen. Violation Type und Remove Entry enthalten allerdings nur wenig Bezug zur realen Welt. Der Update- und der Clear-Button hingegen haben Analogiepotential.

- **Benutzerkontrolle und -freiheit**: Ist der Benutzer stets in der Lage, das System zu kontrollieren?

Diese Heuristik ist nur im geringen Grad auf das GUI anwendbar, da nur das Frame und der Tabelleninhalt gezeigt werden. Durch das 'X'-Symbol des Frames kann der Benutzer die Anwendung jederzeit schließen.

- **Konsistenz und Standards**: Wurde bei der Erstellung der Schnittstelle darauf geachtet, stets dieselbe Formulierung/dieselbe Modellierungsform zu verwenden?

Auch diese Heuristik kann wie die vorige nur in einem geringen Ausmaß angewandt werden, da der Benutzer nur einen Frame sieht. Allerdings wird innerhalb dieses Frames stets dieselbe Art der Formulierung verwendet, um die Tabelleneinträge zu erstellen. Auch die Form und die Beschriftung der Buttons ist stets gleich.

- **Fehlervermeidung**: Wurde die Schnittstelle so designed, dass Fehler vermieden werden?

Es gibt keine intuitiven Rückschlüsse, wie das Interface Fehler vermeiden könnte.

- **Wiedererkennen, statt sich erinnern**: Wird dem User durch die Anzeige kognitive Last genommen?

Durch die Tabellenüberschrift kann eine Verbindung zu Graphen, die durch Nachrichtenaustausch interagieren, hergestellt werden. Aber auch aus der Tabelle kann die Schlussfolgerung gezogen werden, dass die Tabelle durch die Buttons verändert werden kann. Des weiteren ist die Beschriftung der Buttons eindeutig, so dass die Funktionalität dieser klar bestimmt ist und so der Arbeitsaufwand reduziert wird.

- **Flexibilität und Effizienz der Benutzung**: Ist das System editierbar, sodass erfahrene User die Effizienz steigern können?

Individuelle Anpassungen sind nicht ersichtlich.

- **Ästhetik und minimalistisches Design**: Wurden die Elemente der Schnittstelle auf ihre Funktion reduziert und wurde dennoch die Funktionalität durch das Design unterstützt?

Es gibt keine unnötigen Elemente in der Schnittstelle. Alle abgebildeten Elemente haben eine Bedeutung. Durch das Verwenden einer Tabelle wird deutlich, dass die Einträge anhand von bestimmten Charakteristiken definiert werden. Weiterhin hat eine Tabelle die Eigenschaft, viele Elemente übersichtlich anzuordnen.

- **Hilfe beim Erkennen, Diagnostizieren und Beheben von Fehlern**: Wird beim Auftreten von Fehlern Hilfestellung gegeben?

Nein.

- **Hilfe und Dokumentation**: Gibt es ein Handbuch?

Nein.

5.1.2 ISO 9241-110

Neben den allgemein anerkannten Prinzipien Nielsens wurden die Anforderungen an eine Mensch-Maschine-Schnittstelle in der ISO 9241 -110 aufgeführt[10]. Diese Norm bewertet die Benutzerschnittstelle unter der Betrachtung folgender Kriterien.

- **Aufgabenangemessenheit**: Können durch die Funktionen des Systems alle geforderten Ziele erreicht werden?

Das Plugin ist in der Lage, alle Inkonsistenzen eines SMLRuntimeStateGraph zu ermitteln und in einer Tabelle anzuzeigen. Weiterhin können die Inkonsistenzen abgespeichert und anschließend geladen werden. Dadurch können neue Inkonsistenzen mit denen der vorigen Spezifikation verglichen und diese als neu oder wiederkehrend charakterisiert werden.

- **Selbstbeschreibungsfähigkeit**: Kann der Benutzer sagen, an welcher Stelle des Systems er sich befindet, welche Aktionen durchgeführt werden können und welche Auswirkung diese haben?

Ähnlich wie bei der Heuristik 'Sichtbarkeit des Systemstatus' ist der Benutzer durch die Interpretation der Tabellenbeschriftung in der Lage, den Dialog einer Stelle im System zuzuordnen. Durch die Beschriftung der Buttons kann der Benutzer erkennen, welche Aktionen durchgeführt werden können. Die 'Remove Entry?'-Spalte kann ohne weitere Erklärung so aufgefasst werden, als dass der zugehörige Eintrag einer Checkbox durch die Auswahl dieser gelöscht wird. Auch das Textfeld gibt einen Hinweis darauf, dass die Tabelle in Kombination mit den Lade- und Speicherbuttons Inhalte laden und speichern kann.

- **Steuerbarkeit**: Ist der Benutzer in der Lage, die Richtung und die Geschwindkeit des Dialogs zu beeinflussen? Können Unterbrechungen und Rückschritte angewandt werden?

Der Dialog gibt keine Richtung vor. Weiterhin ist der Nutzer nicht an Abfolgen gebunden und kann somit die Geschwindigkeit bestimmen. Rückschritte sind nur begrenzt möglich, indem Tabelleninhalte erneut geladen werden können.

- **Erwartungskonformität**: Ist die Bedienbarkeit einheitlich? Kann vorhergesehen werden, wie lange die Ausführung bestimmter Aktionen benötigt? Ist die Anordnung der Elemente einheitlich gestaltet?

Die Bedienbarkeit ist einheitlich. Das Drücken eines Knopfes löst in allen Fällen eine Manipulation des Tabelleninhaltes aus. Auch die Anordnung ist einheitlich gestaltet. Die Ausführungszeit befindet sich bei allen Aktionen im Bereich von Millisekunden, sodass dieser Punkt außer acht gelassen werden kann.

- **Fehlertoleranz**: Bleiben Eigenschaften und Funktion bei unvorhergesehenen Fehlern in der Eingabe erhalten?

Der Input erfolgt stets in Form eines SMLRuntimeStateGraphs. Daher ist es nicht möglich, Eigenschaften oder Funktionen durch eine Eingabe zu verändern.

- **Individualisierbarkeit**: Kann das System auf den Benutzer abgestimmt werden?

Eine Individualisierung ist, wie in den Heuristiken erwähnt, nicht möglich. Das Plugin konzentriert sich auf die Bereitstellung von Informationen aus Datenstrukturen, die klar definiert sind.

- **Lernförderlichkeit**: Unterstützt die Anwendung den Benutzer beim Erlernen der Nutzung?

Da sich die Funktionalität des Plugins auf wenige Methoden begrenzt, ist der Umfang der erlernbaren Aspekte begrenzt. Nach wenigen Anwendungen ist der Benutzer bereits in der Lage, die Verwendung ohne Hilfestellung oder Erklärungen durchführen zu können.

5.2 Umfrage

Um die Evaluation neben der theoretischen Auswertung durch empirische Daten zu stützen, wird durch eine Umfrage unter uninformierten Personen die Bewertung des Interfaces erweitert. Die Umfrage konzentriert sich auf die Bewertung des GUI anhand anwendbarer Heuristiken Nielsens und Merkmalen der ISO 9241-110. Dabei wird die Umfrage anhand der Methodik von Hegner[10] durchgeführt, um eine subjektive Meinung bezüglich der Erfüllung der angegeben Kriterien zu erhalten. Der Fragebogen ist dem Anhang beigefügt. Dabei ist anzumerken, dass die Erweiterung des Textfeldes zur Eingabe von Dateinamen erst nach der Umfrage entstanden ist.

5.2.1 Ergebnis der Auswertung

Nachdem die Umfrage auf dem Universitätscampus durchgeführt wurde, ergab sich das folgende Ergebnis.

In Abbildung 5.1 ist die Auswertung der Umfrage abgebildet. Es wurden zwanzig Personen befragt, die im Durchschnitt 22 Jahre alt sind. Alle befragten Personen sind Studenten der Informatik und arbeiten regelmäßig mit Graphical User Interfaces. 85 Prozent der befragten Personen sind männlich. Die durchschnittliche Bewertung beläuft sich auf 2,2, was gleichzusetzen mit der Bewertung 'trifft geringfügig zu' ist. Die ersten sechs Kriterien wurden im Durchschnitt mit 2,9 bewertet, was der Bewertung 'trifft teilweise zu'

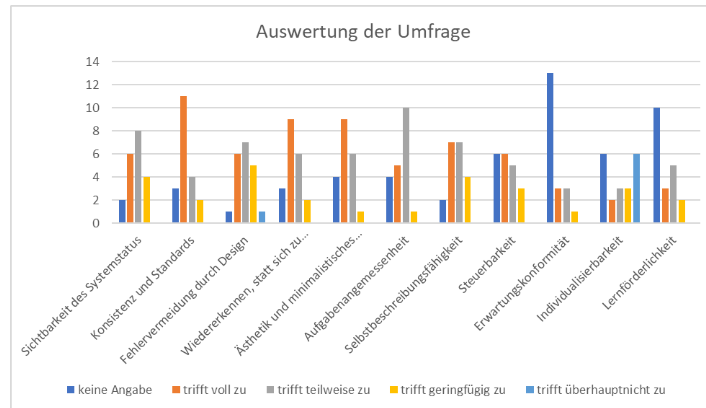

Abbildung 5.1: Auswertung der Umfrage

entspricht. Die letzten fünf Kriterien im Durchschnitt nur mit 1,85('trifft geringfügig zu'). Auffällig dabei ist, dass die ersten sechs Kriterien die Heurisitken Nielsens[10] und die letzten fünf Kriterien die Anforderungen der ISO 9241 sind. Während die Testpersonen die Evaluationsfragebögen ausfüllten, hat sich durch Nachfragen herausgestellt, dass die Kriterien, die statische Elemente des GUI enthielten, verstanden und gut bewertet werden konnten, die dynamischen Elemente, wie *Erwartungkonformität* oder *Lernbarkeit*, allerdings im Kontext nicht verstanden und oft mit 'keine Angabe' bewertet wurden. Daraus lässt sich der Unterschied zwischen den Evaluationskriterien erklären. Weiterhin kann daraus geschlossen werden, dass diese Kriterien eine genauere Beschreibung, als die, die auf dem Evaluationsfragebogen vermerkt war, benötigen, um bewertet zu werden.

5.3 Evaluation von Funktionalität und Usability

5.3.1 Cognitive Walkthrough

Der Cognitive Walkthrough gehört zu den analytischen Evaluationsverfahren. Er wird in der Regel von Usability-Experten durchgeführt. Dabei versetzt sich der Anwender in die Lage der Zielperson und durchläuft die Funktionalitäten der Anwendung. Ziel des Cognitive Walkthroughs ist es, herauszufinden, wo und warum das Design die Kommunikation zwischen User und Interface beeinträchtigen könnte. Im Folgenden werden die Bestandteile des Cognitive Walkthroughs anhand eines Beispiels erklärt. Die Testperson ist ein Diplom-Mathematiker, der in der Software-Branche arbeitet.

- **Phase 1: Vorbereitungsphase**:

  Benutzercharakteristiken/Zielgruppe:

ScenarioTools wird hauptsächlich von Studenten und Angehörigen des Instituts für Software Engineering der Leibniz Universität Hannover verwendet. Vertiefend handelt es sich bei einem Großteil dieser Personen um Software Ingenieure, die eine fundierte Ausbildung besitzen und Erfahrungen in dem Umgang mit Entwicklungstools haben.

Beispielaufgaben:

Der imaginäre Ingenieur soll nach dem Anzeigen der initialen Inkonsistenzen, zuerst den Inhalt der abgespeicherten Datei laden, dann das erste Element aus der Tabelle entfernen, daraufhin den ersten Eintrag mit dem Kommentar 'test' versehen, anschließend die dadurch entstandene Tabelle abspeichern und abschließend den Tabelleninhalt löschen und die abgespeicherten Inhalte laden.

Beschreibung des Interfaces:

Nachdem der Ingenieur die abgespeicherten Daten geladen hat, werden diese den anfänglichen Informationen hinzugefügt. Durch das Entfernen eines Eintrages durch die 'Remove Entry'- 'Update' Folge, wird die Tabelle um einen Eintrag reduziert und die Anordnung der Einträge sortiert. Das Hinzufügen des Kommentars ändert nur die Zelle, die beschrieben wird. Ähnlich führt das Abspeichern zu keiner Änderung der Anzeige. Durch das Löschen wird der Tabelleninhalt geleert und mit einem anschließenden Laden mit den zuvor gelöschten Daten gefüllt.

Handlungssequenzen:

Das Laden der abgespeicherten Dateien erfolgt durch die Betätigung des 'Load Data'- Buttons. Das Markieren der Combobox des ersten Eintrages und dem anschließenden Betätigen des 'Update'-Buttons, resultiert in dem Löschen der ersten Zeile und dem Sortieren der Tabelleninhalte. Durch die Auswahl der 'Comments' Zelle des ersten Eintrages kann diese beschrieben und 'test' eingefügt werden. Eine anschließende Betätigung des 'Safe'-Buttons speichert den Tabelleninhalt in eine externe comma seperated value(CSV) Datei ab. Der 'Clear'-Button entfernt den Tabelleninhalt, und durch die Betätigung des 'Load'- Buttons wird die Tabelle wieder mit den zuvor abgespeicherten Inhalten gefüllt.

- **Analyse**

  Die Testperson hat, nachdem ihr das Setup erklärt wurde, den ersten Schritt spontan korrekt durchgeführt. Dabei hat sie sich bei der Erklärung der Handlung auf die Beschriftung des 'Load' - Buttons bezogen. Den zweiten Schritt hat die Testperson richtig angefangen, aber nicht korrekt zu Ende geführt. Nachdem sie die Checkbox ausgewählt hat, hat sie anfangs erst kurz gezögert und sich gefragt,

warum der Eintrag nicht gelöscht wird. Nach einer kurzen Pause kam sie auf die richtige Lösung und betätigte den Update-Button. Den dritten Schritt hat die Testperson nicht korrekt durchführen können, weil die Editierfunktion der Spalte nicht intuitiv ersichtlich war. Durch die zufällige Auswahl eines Feldes der Spalte entdeckte die Testperson, dass sich dieses Feld editieren lässt. Somit konnte sie die Aufgabe durchführen. Das Abspeichern und das Löschen der Tabelleninhalte konnte sie durch die Beschriftung der Buttons intuitiv erkennen.

- **Protokollierung kritischer Informationen**

  Das Editieren der Kommentare hat auf dem intuitive Wege nicht funktioniert. Aufgrund des Fehlens eines Edit-Buttons war es für die Testperson nicht klar ersichtlich, wie sie das Editieren der Spalte durchführen kann.

- **Revision des Interfaces**

  Die Anordnung und die Beschriftung der Buttons hat dazu beigetragen, einen Großteil der Aufgaben intuitiv durchzuführen. Negativ aufzuführen ist, dass die Oberfläche keinen Hinweis darauf gibt, wie das Editieren der Kommentare durchzuführen ist.

5.3.2 Usability

Usability wird im 11. Teil der ISO 9241 als das Produkt aus Effektivität, Effizienz und Zufriedenstellung definiert[10]. Die Evaluation der Usability impliziert also die Analyse dieser Faktoren bezogen auf die Umsetzung im Plugin. Dazu werden nacheinander Effektivität, Effizienz und Zufriedenstellung des Plugins analysiert.

Effektivität

Effektivität ist eine Maßeinheit, die angibt, wie sehr ein Ergebnis dem vorgegebenen Ziel entspricht. Dabei ist die Wahl der Vergleichsgröße entscheidend dafür, wie die Effektivität gemessen wird. Wenn das Finden aller Inkonsistenzen als Kriterium für die Effektivität verwendet wird, so ist das Plugin insofern effektiv, als dass der Algorithmus alle Inkonsistenzen des übergebenen SMLRuntimeStateGraphes finden kann. Auch die Anzeige der Inkonsistenzen ist effektiv, da alle gefundenen Inkonsistenzen in einer Tabelle angezeigt werden können. Nimmt man nun das Ändern einer SML-Spezifikation als Effektivitätsmaß, so ist es nicht möglich eine Aussage zu treffen, da das Plugin nur Hilfestellung zur Behebung von Inkonsistenzen gibt, nicht aber das Beheben der Inkonsistenzen durchführt.

Effizienz

Effizienz beschreibt den Unterschied zwischen dem Ergebnis eines Tests vor und nach einer Veränderung des Systems. Bezogen auf das Plugin bedeutet Effektivität also, wie oder ob durch das Plugin eine Steigerung des Nutzens zustande gekommen ist. Die Steigerung ist ein abstraktes Maß und kann verschiedene Werte annehmen. In diesem Fall ist der Faktor Zeit ein gut gewähltes Vergleichskriterium, da das Suchen der Inkonsistenzen in einem Zustandsgraphen und die Zuordnung dieser beziehungsweise das Erkennen wiederkehrender Inkonsistenzen Zeit beansprucht.

Zufriedenstellung

Die Handhabung von Systemen unterscheidet sich durch die Art und Weise der Benutzung. Ist ein System durch gute Platzierung von Elementen übersichtlich, wird es von Usern bevorzugt verwendet, weil sie sich eher an einfache Benutzungen erinnern als an komplizierte. Die Zufriedenstellung wird dabei durch die Faktoren Spaß und Intuitivität beeinflusst. Die Intuitivität der Benutzung wird durch die Anordnung der Elemente und den Beschriftungen der Tabellenköpfe gefördert. Wie im Cognitive Walkthrough gezeigt, kann ein Großteil der Aufgaben beim ersten Mal durchgeführt werden. Der Spaß wird durch das Einsparen von Zeit bei der Suche nach Inkonsistenzen und dem Vergleich dieser auf wiederkehrende Elemente erzeugt.

5.3.3 Experiment

Ein geeignetes Experiment zum Überprüfen der Effizienzsteigerung ist die Durchführung einer Spezifikationserweiterungen mit vorprogrammiertem Auftreten von Inkonsistenzen unter Verwendung des Plugins und ohne Verwendung des Plugins. Es wird dabei angenommen, dass die Probanten keine Information über die Lösung der Aufgabe haben. Um eine Effizienzsteigerung festzustellen, bedarf es der Erhebung empirischer Daten unter der Verwendung neutraler Testpersonen.

Experimentvorbereitung

Der Computer ist angeschaltet und die Entwicklungsumgebung von ScnenarioTools ist geöffnet. Die zu integrierende Anforderung an das bestehende System ist textuell auf einem Blatt beschrieben(im Anhang zu finden). Eine Stoppuhr steht bereit und wird bei Beginn der Umsetzung aktiviert. Ist die Testperson fertig, wird die Zeit gestoppt und das Resultat vermerkt.

Experimentdurchführung

Person A und Person B befinden sich in getrennten Räumen. Dann wird Person A in den Experimentraum gebeten und führt die Modifizierung ohne die Verwendung des Plugins durch. Dabei wird die Zeit festgehalten und das Resultat gespeichert. Anschließend verlässt Person A den Experimentraum und Person B betritt diesen. Person B führt dann die Modifizierung mit der Benutzung des Plugins durch. Auch dieses Mal wird notiert, wie lange Person B für die Bearbeitung gebraucht hat und wie das Resultat aussieht. Im Anschluss verlässt Person B den Experimentraum.

Experimentbeobachtungen

Die Durchführung des Experiments konnte nicht durchgeführt werden, weil keine Testpersonen gefunden werden konnten.

Fazit des Experiments

Das vorgestellte Verfahren stellt eine Modell dar, wie die Effizienz des Plugins berechnet werden kann.

5.3.4 SOLL - IST Analyse

Die Funktionalität eines Systems kann durch die Erfüllung ihrer Soll-Anforderungen überprüft werden. Dabei werden diskrete Werte angenommen. Bezogen auf das Plugin bedeutet dies ein Abgleich der gestellten Anforderungen mit den implementierten Funktionalitäten.

Die im dritten Kapitel formulierten Anforderungen lauteten:

a) SOLL: Inkonsistenzen müssen anhand von fest definierten Parametern charakterisiert werden können

IST: Durch die Abarbeitung aller Pfade des SMLRuntimeStateGraphen und dem Extrahieren der in der Implementierung gewählten Kriterien in der *getInconsistencies()*-Methode wird gewährleistet, dass die Charakterisierung von Inkonsistenzen erfüllt wird.

b) SOLL: Inkonsistenzen müssen übersichtlich angezeigt werden können

IST: Das Anzeigen der Inkonsistenzen erfolgt in Form einer Tabelle durch das Hinzufügen der extrahierten Werte des SMLRuntimeStateGraphs in eine ArrayListe, nachdem sie in Objekte des Typs Model umgeformt wurden, wodurch die Anforderung erfüllt wird.

c) SOLL: Inkonsistenzen müssen anhand von Parametern gruppiert werden können

IST: Durch den Aufruf der *.sort(ArrayList<Model>)* Funktion aus der Methods- Klasse werden Model-Objekte innerhalb einer Liste nach dem Kriterium Critical Message alphabetisch sortiert, wodurch die Anforderung erfüllt wird.

d) SOLL: Inkonsistenzen müssen abgespeichert und geladen werden können

IST: Mithilfe des FileReader und des FileWriters der Utility Library kann, wie Kapitel 4 beschrieben, der Tabelleninhalt gespeichert und geladen werden, wodurch die Anforderung erfüllt wird.

e) SOLL: Die Anzeige muss manuell manipuliert werden können, um eigene Gedanken in die Anzeige fließen zu lassen

IST: Die Spalte 'Comments' wurde dem Protoypen hinzugefügt, die mithilfe der *isCellEditable()*- Funktion des DefaultTableModels editierbar gemacht wurden, sodass diese manuell über die Tastatur beschrieben werden können. Die Formatierung wird dadurch nicht beeinflusst. Einzig der Platz ist begrenzt, sodass im allgemeinen nur Stichworte vermerkt werden können. Weiterhin ermöglicht der 'Update'- Button des GUI das Entfernen ausgewählter Einträge und der 'Clear'- Button die Löschung aller Elemente der Tabelle.

5.4 Zusammenfassung der Evaluation

Das Plugin erfüllt auf der visuellen Ebene allgemein anerkannte Normen und Prinzipien. Auch auf der funktionalen Ebene wurden die Anforderungen an das Plugin erfüllt. Des weiteren wurde durch eine Umfrage und einem Cognitive Walkthrough gezeigt, dass viele Funktionen des Plugins intuitiv richtig umgesetzt wurden. Das Experiment, das die Effiziensteigerung beweisen könnte, konnte aufgrund der geringen potentiellen Testpersonen nicht durchgeführt werden.

Kapitel 6

Abgrenzung verwandter Arbeiten

In diesem Kapitel findet eine Abgrenzung von Arbeiten, die sich ebenfalls mit dem Finden von Fehlern innerhalb eines Prozesses beschäftigen, statt. Dazu wird zunächst der einschließende Oberbegriff der Arbeit- das Debugging- aufgeführt und erklärt. Anschließend werden zentrale Aussagen der Arbeit zusammengefasst, um zu zeigen, wie die Arbeit in das Debugging und den Zyklus des Software-Engineerings eingeordnet werden kann. Abschließend werden verwandte Arbeiten und ihre zentralen Aussagen aufgeführt, um zu zeigen, wie das Debugging in anderen Bereichen verwendet werden kann.

6.1 Debugging

6.1.1 Debugging

Das Debugging behandelt das Identifizieren und Entfernen von *Bugs* in Systemen. Ein *Bug* beschreibt einen Fehler in einem System, der wiederum als Unterschied zwischen der Funktionalität, die ausgeführt wird, und der Funktionalität, die ausgeführt werden soll, definiert ist. Im Debugging geht es darum, den Ursprung des Bugs zu identifizieren. Im Folgenden wird dann der Fehler behoben und das System erneut getestet. Daher setzt das Debugging Wissen bzgl. der Funktionalität voraus und ist eine der kompliziertesten Software-Skills[22].

6.1.2 Einordnen der Arbeit

Die Entwicklung des Plugins hat gezeigt, wie das Debugging gezielt eingesetzt werden kann, um in Scenarios Inkonsistenzen zu finden und zu charakterisieren. Wie in der ersten Phase des Debuggings geht es darum, Fehlerquellen in der Formulierung von Szenario- basierten Modellen zu identifizieren und für weitere Bearbeitung übersichtlich bereitzustellen. Diese

Arbeit konzentriert sich darauf, nicht realisierbare Spezifikationen auf den Ursprung der Fehlerquelle zu untersuchen, um dem angehenden Ingenieur aufzuzeigen, wo er die Spezifikation ändern muss. Ziel dabei ist es ein, besseres Verständnis des Fehlers zu erhalten.

6.2 Abgrenzung zu ähnlichen Arbeiten

Ähnlich zu der im vorigen Kapitel beschriebenen Thematik, beschäftigt sich die Bachelorarbeit von Gutjahr[8] mit der Simulation von unrealisierbaren Szenario-basierten Spezifikationen. Gutjahr erweiterte den Synthese-Algorithmus, der bei einer großen Spezifikation nicht einfach zu verstehen war, um die Gegenstrategie, die dem Benutzer aufzeigt, wie die Umwelt einen Widerspruch herbeiführt. Eine weitere Arbeit ist die Masterarbeit von Schwind[21], die das Modellbasierte Debugging behandelt. Schwind erklärt in seiner Arbeit den Wandel der Modellierung von Software- Systemen und führt auf, warum die Modellierung reaktiver Systeme an Bedeutung gewinnt, indem er einen Prototypen reaktive eingebettete Systeme konzipiert und implementiert. Des weiteren lassen sich in den Arbeiten von Marron[13] und Moaz[14] viele Ansätze finden, die das Verständnisproblem von Inkonsistenzen innerhalb des Debebugginprozesses beschreiben. In der Arbeit 'Counter Play-Out: Executing Unrealizable Scenario-Based Specifications' von Moaz und Sa'ar[15] z.B. wird eine interaktive Debugging-Methodik entwickelt, die den Ursprung der Inkonsistenz einer unrealisierbaren Spezifikation durch einen Play- Out- Algorithmus findet. Um eine weitere Arbeit aufzuzählen, wird in 'GR(1) Synthesis for LTL Specification Patterns' von Moaz und Ringert[16] gezeigt, wie das Formulieren von Spezifikationen durch den Einsatz von Industriemustern vereinfacht werden kann. Dazu werden die 55 Designpatterns in den Syntheseprozess integriert, um die Synthese von Spezifikationen, die mit den ausgewählten Patterns formuliert wurden, zu automatisieren.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

7.1.1 Inhaltliche Zusammenfassung

In dieser Arbeit ist ein Plugin entstanden, welches das Debugging von domänenspezifischen Spezifikationen unterstützt. Nachdem in den ersten zwei Kapiteln in die Thematik eingeführt und essentielle Grundlagen vermittelt wurden, entstand das Plugin im dritten und vierten Kapitel der Arbeit. Die anschließende Evaluation im fünften Kapitel zeigt, wie das Plugin visuelle und funktionale Anforderungen erfüllt. Während der abschließenden Abgrenzung im sechsten Kapitel der Arbeit wurde eine Kategorisierung der Arbeit in übergeordnete Bereiche des Software-Engineerings vorgenommen.

7.1.2 Funktionale Zusammenfassung

Das Plugin kann Inkonsistenzen anhand von wohl überlegten Kriterien charakterisieren und übersichtlich in Form einer Tabelle anzeigen. Dadurch wird dem Benutzer die Suche nach Inkonsistenzen und die Einordnung derer erleichtert. Weiterhin kann durch das Plugin gewährleistet werden, dass Inkonsistenzen, die schon behandelt wurden, wiedererkannt werden. Des weiteren enthält das Plugin die Funktionalität, jede Inkonsistenz mit eigenen Kommentare zu versehen. Durch die Möglichkeit, Inkonsistenzen abzuspeichern und zu einem späteren Zeitpunkt laden zu können, kann in Unternehmen, die mit ScenarioTools arbeiten, inkrementell gearbeitet werden, ohne den Arbeitsfortschritt verwerfen zu müssen.

7.2 Ausblick

Es stellt sich hier die Frage, ob die Entwicklung durch die Verwendung des Plugins effizienter gestaltet werden kann. Das Experiment, welches während der Evaluation beschrieben, aber aufgrund von einer geringen Anzahl geeigneter Testpersonen nicht aussagekräftig ausgeführt werden konnte, könnte einen Aufschluss darüber geben.

Des weiteren öffnet die Arbeit Türen für Arbeiten, die auf dieser Arbeit aufbauen. In folgenden Arbeiten z.B. könnte die Entwicklung von Verfahren, die sich auf die generierten Daten des Plugins berufen und das Beheben der Inkonsistenzen in der SML- Spezifikation durch einen Algorithmus in Anlehnung an die Syntheseautomatisierung durch Mustereinsatz von Moaz[16] vereinfachen, behandelt werden.

Anhang A

Ein Anhang

Evaluationsfragebogen

Persönliche Angaben

Alter:
Geschlecht:

Studiengang:
Datum:

Richtlinien

Füllen Sie den Bewertungsbogen unter Verwendung des folgenden Schlüssels aus:

1 = *trifft überhaupt nicht zu*
2 = *trifft geringfügig zu*
3 = *trifft teilweise zu*
4 = *trifft voll zu*
5 = *keine Angabe*

Anforderungen und Heuristiken

	(5)	(4)	(3)	(2)	(1)
Sichtbarkeit des Systemstatus	☐	☐	☐	☐	☐
Konsistenz und Standards	☐	☐	☐	☐	☐
Fehlervermeidung durch Design	☐	☐	☐	☐	☐
Wiedererkennen, statt sich zu erinnern	☐	☐	☐	☐	☐
Ästhetik und minimalistisches Design	☐	☐	☐	☐	☐
Aufgabenangemessenheit	☐	☐	☐	☐	☐
Selbstbeschreibungsfähigkeit	☐	☐	☐	☐	☐
Steuerbarkeit	☐	☐	☐	☐	☐
Erwartungskonformität	☐	☐	☐	☐	☐
Individualisierbarkeit	☐	☐	☐	☐	☐
Lernförderlichkeit	☐	☐	☐	☐	☐

Beschreibung der Anforderungen und Heuristiken:

Sichtbarkeit des Systemstatus:	Das System gibt Feedback über aktuelle Vorgänge.
Konsistenz und Standards:	Werden dieselben Wörter/Designelemente verwendet?
Fehlervermeidung durch Design:	Das Design verringert die Fehlerquote.
Wiedererkennen, statt sich zu erinnern:	Das Design verringert den Arbeitsaufwand.
Ästhetik und min. Design:	Jedes Element hat einen Nutzen.
Aufgabenangemessenheit:	Reichen die Elemente aus, die gewünschte Funktionalität bereitzustellen?
Selbstbeschreibungsfähigkeit:	Die Dialogelemente können ohne zusätzliche Erklärung verstanden werden.
Steuerbarkeit:	Der Benutzer hat die Kontrolle über Richtung und Geschwindigkeit des Dialogablaufs
Erwartungskonformität:	Die Aktionen werden wie erwartet ausgeführt.
Individualisierbarkeit:	Die Funktionen des Dialogs können an den Benutzer angepasst werden.
Lernförderlichkeit:	Der Dialog unterstützt den Benutzer beim Erlernen der bereitgestellten Funktionen.

Beschreibung der Funktionalität:

Abgebildet ist das GUI eines Plugins. Bei Ausführung werden Zustände eines Graphen extrahiert, sobald sie eine bestimmte Eigenschaft aufweisen und in der Tabelle angezeigt. Die Kopfzeile der Tabelle gibt an, welche Informationen die Spalten anzeigen.

Die Buttons lösen folgende Funktionen aus:

Wenn der Update-Button gedrückt wird, werden alle Einträge, deren Checkbox markiert ist, entfernt, und die Einträge in der Tabelle werden der Spalte Critical Message nach sortiert.

Wenn der Clear-Button gedrückt wird, wird der gesamten Inhalt der Tabelle gelöscht.

Der Load-Button lädt abgespeicherte Daten

Der Safe-Button speichert die aktuellen Tabelleninhalte in eine Datei.

Durch die Auswahl einer Zeile mit dem Cursor kann diese editiert werden.

Das Berühren des Cursors mit der Kopfzeile zeigt einen Tooltip an.

Inconsistency Table

Path	Critical Message	Alternative Messages	Violation Type	Remove Entry?
1, 10, 11, 12, 13, 14	controller->coffeeMachine.pourCoffee()	controller->controller.setcupDetected(true)	Safety Violation	☐
14, 14			Liveness Violation	☐

Path from the root to the violated State in Indexes of the states

Update | Clear | Load data | Safe data

Literaturverzeichnis

[1] H. Balzert. *Lehrbuch der Software-Technik. Bd.1. Software-Entwicklung*, volume 1. Spektrum Akademischer Verlag, 2001.

[2] P. E. Ceruzzi. *A Histrory of Modern Computing*, volume 1. MIT Press, 2003.

[3] J. D. Elizabeth Hull, Ken Jackson. *Requirements Engineering*, volume 3. Springer Verlag, 2011.

[4] B. Fadel and M. B. Trilling. *Four-Dimensional Education: The Competencies Learners Need to Succeed*, volume 1. CreateSpace Independent Publishing Platform, 2015.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, volume 1. Addison-Wesley Professional Computing Series, 1994.

[6] J. Greenyer and et al. Homepage von scenariotools. *www.scenarioTools.org*, 2017.

[7] 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(Supplement C):15 – 27, 2017. Special Issue on MODELS'16.

[8] T. Gutjahr. Scenariotools counter-play-out simulation zur analyse von unrealisierbaren szenariobasierten spezifikationen. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2014.

[9] Harel and Marelly. *Come, Let's Play Scenario-Based Programming Using LSCs and the Play-Engine*, volume 1. Springer-Verlag Berlin und Heidelberg GmbH und Co. KG, 2012.

[10] M. Hegener. *Methoden zur Evaluation von Software*, volume 1. Informationszentrum Sozialwissenschaften der Arbeitsgemeinschaft Sozialwissenschaftlicher Institute e.V.(ASI), 2003.

[11] B. Kadioglu. *Unterstützung der Informationsflussanalyse mit einer domänenspezifischen Sprache (DSL).* PhD thesis, Leibniz Universität Hannover, Hannover, October 2017.

[12] M. Koegel and J. Helming. What every eclipse developer should know about emf. *eclipsesource.com/blogs/tutorials/emf-tutorial*, 2017.

[13] Marron. Publication. Website, December 2017.

[14] Moaz. Publications. Website, December 2017.

[15] S. Moaz and Y. S. ar. Counter play-out: Executing unrealizable scenario-based specifications. *http://www.cs.tau.ac.il/ maozs/papers/counterplayout-icse13.pdf*, 2013.

[16] S. Moaz and J. O. Ringert. Gr(1) synthesis for ltl specification patterns. *http://www.cs.tau.ac.il/ maozs/papers/syntech-patterns-fse15.pdf*, 2015.

[17] H. Mössenböck. *Sprechen Sie Java?: Eine Einführung in das systematische Programmieren.* dpunkt-Lehrbuch. Dpunkt.Verlag GmbH, 2014.

[18] Oracle. How to use tables. Website, December 2017.

[19] A. Reinefeld. *Spielbaum-Suchverfahren.* Springer Verlag, 2000.

[20] W. Royce. Managing the development of large software systems. Website, December 2017.

[21] S. Schwind. *Modellbasiertes Debugging - Konzeption und Implementierung eines Werkzeugprototypen für Reaktive Eingebettet Systeme.* PhD thesis, Technische Universität München, München, March 2010.

[22] S. Stephen. *Robust Java: Exception Handling, Testing and Debugging*, volume 1. Prentice Hall, 2005.

[23] vogella GmbH. Einführung in die programmierung mit eclipse. *http://www.vogella.com/tutorials/Eclipse/article.html*, 2017.