Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 282

Entwicklung eines STPA-Verifiers als Eclipse-Plug-in für die Verifikation von Software-Sicherheitsanforderungen

Lukas Balzer

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	M. Sc. Asim Abdulkhaleq
Beginn am:	2015-10-20
Beendet am:	2016-04-20
CR-Nummer:	D.2.4, I.6.4, F.1.1, F.4.1

Kurzfassung

Um die Sicherheit in kritischen Softwaresystemen zu gewährleisten ist immer häufiger eine Verifikation der Software in einem Systemkontext notwendig. Hierfür ist in den letzten Jahren die Verifikation von Softwaresystemen durch Model Checking, bedingt durch die wachsende Anzahl an dafür zur Verfügung stehenden Werkzeugen, eine bewährte Methode geworden. Diese Arbeit stellt auf Grundlage des in STPA SwISS [AWL15] vorgestellten Konzeptes eine Software zur automatisierten Ausführung von LTL und CTL Model Checking mit den Werkzeugen Spin und NuSMV bietet. Dabei können die Sicherheitsanforderungen sowohl manuell eingegeben werden, als auch aus einer STPA Analyse importiert werden. Das Ergebnis dieser Arbeit soll ein Ansatz zur Kombination einer Gefahrenanalyse auf Systemebene und einer Verifikation dieses Systems auf Implementierungsebene sein. Zu diesem Zweck wird der STPA Verifier zur automatisierten Verifikation von Sicherheitsanforderungen und Dokumentation der Ergebnisse vorgestellt.

Abstract

To verify the safety on a critical softwaresystem includes more and more often the task of verifying Software in the context of the system. In the last years verifying software by using formal model checking has become a more and more popular method due to the increasing number of available tool support. This work presents a Software based on the concepts of the STPA SwISS approach [AWL15] that provides a graphical user interface for performing automated LTL and CTL model checking using the Spin or the NuSMV model checker. The safety properties can be derived either manually or by importing the results of a STPA hazard analysis. The result of this work are supposed to be an approach to combine a hazard analysis on system level and a Softwareverification on implementation level. To provide this the STPA Verifier for verfying safety constraints and creating a verification report is presented.

Danksagung

Ich möchte mich vorab bei allen die mich bei der Durchführung meiner Bachelorarbeit und der Entstehung dieses Dokuments unterstützt haben bedanken. Im besonderen möchte ich mich bei Asim Abdulkhaleq bedanken der mich als Betreuer der Bachelorarbeit unterstützt hat. Des weiteren geht mein Dank an Professor Dr. Stefan Wagner für die Betreuung des Themas. Als letztes möchte ich mich noch bei allen Korrektoren dieses Dokuments bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

A-STPA	Automated STPA
BDD	Binary decision diagrams
BMC	Bounded Model Checking
CTL	Computation tree logic
LTL	Linear temporal logic
NNF	Negation Normal Form
NuSMV	A new symbolic model checker
Promela	Process Meta Language
SMV	Symbolic Model Checker
Spin	Simple Promela Interpreter
STPA	System Theoretic Process Analysis
STPA SwISS	STPA for Software-Intensive Systems
STAMP	Systems-Theoretic Accident Model and Processes
XSTAMPP	An eXtensible STAMP Platform
XSTPA	eXtended STPA

1 Einleitung

1.1 Motivation

Software gewinnt immer an mehr Bedeutung in komplexen Systemen und übernimmt zunehmend vitale Funktionen des Systems. Aus diesem Grund ist es immer wichtiger Software nicht nur auf Zuverlässigkeit sondern auch auf Sicherheit im Systemkontext zu prüfen [AW15a]. Viele Gefährdungen und Unfälle in modernen Systemen entstehen oft trotz tadellos funktionierender Software oder gerade auf Grund dieser. In den letzten Jahren wurden die Möglichkeit der automatisierten Softwareverifikation mittels formaler Methoden wie Model Checking immer attraktiver und effizienter durch Softwarelösungen wie den Spin oder NuSMV Model Checker. Doch trotz der immer zuverlässigeren Methoden der System Verifikation und zuverlässigen Gefährdungsanalysen wie STPA besteht immer noch eine Lücke zwischen Sicherheitsanforderungen auf Systemebene die aus einer STPA Analyse resultieren und der Verifikation dieser auf Implementierungsebene durch Formulierung logischer Eigenschaften in einer temporalen Logik.

1.2 Problemstellung

Diese Arbeit nimmt als Grundlage die von Nancy G. Leveson vorgestellte STPA Analyse [Lev11]. Mithilfe dieser Methode lassen sich Systeme auf Gefährdungen und Fehlverhalten hin analysieren wodurch Sicherheitsanforderungen im Systemkontext abgeleitet werden können. Diese Analyse bietet den Vorteil das sie zur Verfeinerung ihres Ergebnisses das Prozess Model der Software ableitet wodurch sich durch Erweiterung des Prozesses [Tho13] schnell konkrete Sicherheitsanforderungen an die Software ableiten lassen. Um diese, in natürlicher Sprache formulierten Sicherheitsanforderungen jedoch zur Verifikation der Software auf Implementierungsebene zu nutzen sind weitere Schritte notwendig. In der Ausarbeitung von Abdukhaleq et al.[AWL15] wird ein umfassender Ansatz zur Verifikation von Software gegen die Ergebnisse einer STPA Gefährdungsanalyse vorgestellt. Dieser sieht den Einsatz sogenannter Model Checker [McM93, Hol07] vor um die zuvor in eine temporale Logik [Muk97] überführten Anforderungen auf Implementierungsebene zu verifizieren.

1.3 Zielsetzung

Ziel dieser Bachelorarbeit ist die Erstellung des STPA Verifiers als Plug-in zur Einbindung formaler Softwareverifikation durch Modellüberprüfung in die schon existierende XSTAMPP Platform. Das Resultat dieser Arbeit soll eine effiziente Überprüfung der Ergebnisse einer STPA Analyse auf Softwareebene ermöglichen sowie die Formulierung und Verwaltung von Sicherheitsanforderungen in LTL(Kapitel 2.3.1) sowie CTL(Kapitel 2.3.2) Syntax.

Eine weitere Anforderung an das Entwickelte Plug-in ist die automatisierte Nutzung des Modex Werkzeugs(Kapitel 2.3.7) und damit die Ableitung von System Modellen auf Implementierungsebene.

Um eine effektive Nutzung des Werkzeugs zu ermöglichen wird eine leicht zu bedienende Oberfläche zur Einbindung und Nutzung der beiden Model Checker Spin (Kapitel 2.3.3) und NuSMV (Kapitel 2.3.5) benötigt. Diese sollte über Komponenten zur Durchführung automatisierter Verifikationssequenzen und Darstellung derer Ergebnisse verfügen.

Als letztes sollte der STPA Verifier über eine Export Funktion verfügen die es erlaubt die Ergebnisse der Verifikation(-en) sowohl in PDF, PNG als auch als CSV Datei abzuspeichern.

1.4 Gliederung

Dieses Dokument ist wie folgt in sechs Kapitel aufgeteilt.

1. **Kapitel 2** stellt die Grundlagen dieser Bachelorarbeit vor. Die Grundlagen sollen eine Einführung in die Terminologie dieser Arbeit geben.
2. **Kapitel 3** zeigt die Analyse der vorgestellten Zielstellung. Im zweiten Teil des Kapitels wird der Entwurf und die Konzeption der Implementierung vorgestellt.
3. **Kapitel 4** präsentiert die Implementierung und die Funktionen des STPA Verifiers.
4. in **Kapitel 5** stellt anhand einer konkreten Anwendung die Funktionsweise des Werkzeugs vor.
5. **Kapitel 6** beinhaltet eine Installationsanleitung sowie eine Liste an Anforderungen und Links um den STPA Verifier nutzen zu können.
6. **Kapitel 7** fasst den Inhalt dieses Dokuments zusammen und schließt dann mit einem Ausblick auf weiterführende Ansätze.

2 Grundlagen

2.1 STAMP

STAMP ist ein von Leveson [Lev04] 2004 vorgestelltes Unfall Model welches Fehlverhalten und dadurch resultierende Unfälle mithilfe theoretischer Betrachtung des Systems analysiert und so versucht einen Unfall auf Interaktionsfehler zwischen bzw. mit den Komponenten zurückzuführen, anstatt auf Ausfälle. STAMP basiert auf der Betrachtung eines Systems durch Betrachtung seiner Sicherheitsregelstruktur (engl.: controlstructure) und der Ableitung von Sicherheitsanforderungen (engl.: safety constraints) die unsichere Regelungen unterbinden. STAMP bietet eine generelle Grundlage für eine systemtheoretische Sicherheitsanalyse.

2.2 STPA

STPA ist eine 2004 von Leveson [Lev11] veröffentlichte Gefährdungsanalyse die eingesetzt werden kann um natürlichsprachliche Sicherheitsanforderungen für ein bestehendes System abzuleiten oder einen Entwicklungsprozess Sicherheitstechnisch zu optimieren. Die Analyse orientiert sich dabei vor allem an den Interaktionen der im System existierenden Komponenten wobei sowohl mechanische Bauteile, Software als auch Menschen als solche betrachtet werden. Durch diese Betrachtungsweise gelingt es nicht nur Fehlfunktionen einzelner Komponenten zu erkennen sondern auch jede Art von gefährdender Interaktion oder Kommunikation.

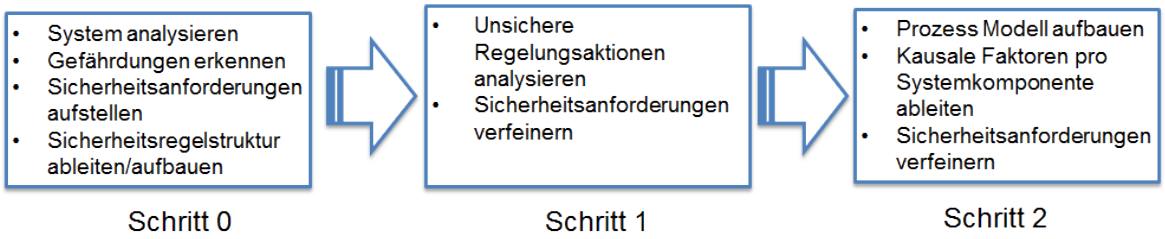


Abbildung 2.1: Schema einer STPA Analyse

Wie bereits erwähnt kann STPA generell für zwei Szenarien eingesetzt werden, einen geführten Entwicklungsprozess oder die Überprüfung/Optimierung eines bestehenden Systems. In beiden Fällen muss vor Beginn der Analyse eine Liste der bestehenden oder gewünschten Systemanforderungen sowie möglichen bekannten Sicherheitslücken oder Unfallszenarien

aufgestellt werden. Des weiteren benötigt STPA als Eingabe eine schematische Darstellung der Sicherheitsregelstruktur des Systems die sämtliche Komponenten sowie deren Kommunikationskanäle untereinander beschreibt. Die darauf folgende eigentliche Analyse umfasst im Wesentlichen zwei Schritte:

1. Im ersten Schritt von STPA werden alle im System möglichen Regelungsaktionen analysiert. Bei Identifikation möglicher unsicherer Regelungsaktionen unterscheidet STPA zwischen vier möglichen Ursachen:
   - Die Gefährdung entsteht durch ein Ausbleiben der Aktion
   - Die Gefährdung entsteht durch die Ausführung der Aktion
   - Die Gefährdung entsteht durch eine zeitlich versetzte Ausführung der Aktion
   - Die Gefährdung entsteht durch eine zu lang oder zu kurz dauernde Ausführung der Aktion

   Durch die Analyse, bezogen auf diese vier Punkte, lässt sich eine Liste an potenziell unsicheren Regelungsaktionen ableiten die sich mittels relativ einfacher grammatikalischer Umformungen in Sicherheitsanforderungen übersetzten lassen.
2. Im zweiten Schritt werden die gerade aufgestellten unsicheren Regelungsaktionen anhand des Softwarecontrollers des Systems betrachtet. Folglich muss die bestehende Darstellung der Sicherheitsregelstruktur um die Prozessmodelle der dem System zugrunde liegenden Abläufe erweitert werden. Das Ziel dieses zweiten Schrittes ist mit Hilfe dieser Erweiterungen die Gültigkeit der abgeleiteten Sicherheitsanforderungen im Systemkontext zu verifizieren und mögliche gefährdenden Abläufe zu analysieren. Dies gelingt durch eine Ursachenanalyse pro Komponente; d.h. eine genaue Untersuchung des bzw. der Prozessmodelle unter Voraussetzung der bis hierhin analysierten Sicherheitsanforderungen.

Erweiterung durch John Thomas

2013 wurde in der Veröffentlichung [Tho13] eine Erweiterung von STPA präsentiert die die in Schritt 2 durchgeführte Ursachenanalyse automatisiert und so versucht den Anteil der identifizierten Risiken im System deutlich zu erhöhen. Dabei werden aus den Prozessmodellen die Prozessvariablen und deren Belegungen ausgelesen und für jede, als sicherheitskritisch befundener Regelungsaktion, in $N * M$ Kontexttabellen mit N = Anzahl der für die Regelungsaktion relevanten Prozessvariablen und N die Anzahl an Wertekombinationen ist.

Control Action	Train Motion	Emergency	Train Position	Hazardous control action?		
				If provided any time in this context	If provided too early in this context	If provided too late in this context
Door open command provided	Train is moving	No emergency	(doesn't matter)	Yes	Yes	Yes
Door open command provided	Train is moving	Emergency exists	(doesn't matter)	Yes	Yes	Yes
Door open command provided	Train is stopped	Emergency exists	(doesn't matter)	No	No	Yes
Door open command provided	Train is stopped	No emergency	Not aligned with platform	Yes	Yes	Yes
Door open command provided	Train is stopped	No emergency	Aligned with platform	No	No	No

Abbildung 2.2: Das Bild zeigt die Kontexttabelle für die *open door* Regelungsaktion in dem in [Tho13] vorgeführten „Train"Beispiel

2.3 Software Sicherheitsverifikation

Sicherheitsverifikation ist der Versuch eine Software von Sicherheitsrisiken zu befreien. Hierfür unterscheidet man generell zwischen zwei Ansätzen. Der erste mit dem sich diese Arbeit nicht tiefer gehend beschäftigen will ist die dynamische Verifikation, also die Verifikation zur Laufzeit einer Software. Eine klassische dynamische Analyse besteht aus Erstellung und Durchführung von Testfällen.
Ein zweiter Ansatz ist die Verifikation mittels formaler Methoden. Diese teilen sich in drei Unterkategorien:

1. Verifikation durch Beweis eines formalen Theorems
2. Durch Deduktion
3. Durch Model Checking
   Der Ansatz des Model Checkings basiert im Wesentlichen auf der abstrakten Modellierung eines System und dem anschließenden Durchlaufen sämtlicher erreichbarer Systemzustände. Model Checking ist eine immer verbreitetere Methode um ein Software intensives System auf formale Anforderungen an Sicherheit und Lebendigkeit zu überprüfen. Um ein System mittels Model Checking zu überprüfen wird ein, in einer passenden Modellsprache wie Promela oder der NuSMV Eingabesprache geschriebenes, Systemmodell benötigt. Dieses wird entweder anhand des Softwaresystems abgeleitet oder wie in Kapitel2.3.7 beschrieben aus einer Quellcode Datei extrahiert.

Abhängig von der Implementierung des benutzten Model Checkers können zwei mögliche Vorgehensweisen zum Einsatz kommen:

3.1. **Explizites Model Checking** transformiert zuerst das gegebene Systemmodell in eine effizient zu durchlaufende Form. Ein Beispiel für ein solches System ist das in Kapitel 2.3.3 vorgestellte Werkzeug *Spin*.

3.2. **Symbolisches Model Checking** basiert wie der Name schon sagt auf einer symbolischen Repräsentation der Zustandsüberänge. Die Idee des symbolischen Model Checkings ist es statt wie im oberen Fall ein Transitionensystem zu entwickeln, lediglich die Zustandsmenge sowie eine Übergangsfunktion abzuspeichern. Der Aufbau dieser Funktion ist vom Aufbau der zu überprüfenden Eigenschaft abhängig [Roz11]. Diese Arbeit beschränkt sich auf Formeln im LTL oder im CTL Syntax. Beim symbolischen CTL Model Checking werden die Zustandsübergänge und die Formeln selbst in binären Entscheidungsdiagrammen (englisch BDD) festgehalten.

2.3.1 Linear Temporal Logic

Sprache und Kalkül um Eigenschaften von Zustandsfolgen auszudrücken, ohne die Zeit explizit zu erwähnen.

H. Peter Gumm [Gum07]

Die lineare temporale Logik kurz LTL ist eine temporale Logik mit deren Hilfe man logische Ausdrücke und deren Gültigkeit über die Zeit formulieren kann. Die Logik erweitert die ihr zugrundeliegende Aussagenlogik durch die Betrachtung in einem zeitlichen Kontext. Dadurch eignet sich die LTL gut um die Gültigkeit von Sicherheitseigenschaften in einer linearen Sequenz von Systemzuständen zu überprüfen. Eine Formel in LTL ist eine Kombination aus logisch verknüpften aussagenlogischen Ausdrücken mit einem der temporalen modalen Operatoren:

1. $\alpha U \beta$ (**U**ntil) um Auszusagen das eine Eigenschaft α gelten muss bis β eintritt
2. $X\ \alpha$(ne**X**t) wenn ein Ausdruck β gelten muss sobald ein anderer α unwahr wird

Wobei auch meistens die folgenden Kombinationen als gegeben angesehen werden.

1. $F\alpha = \text{true} U \alpha$ wenn eine Eigenschaften auf jeden Fall irgendwann wahr werden muss
2. $G\alpha = \neg F \neg \alpha$ oder [] für global gültige Eigenschaften wahr werden muss
3. $\alpha R \beta = \neg(\neg \alpha U \beta)$ wenn ein Ausdruck α gültig sein muss bis einschließlich dem ersten Systemzustand in dem b gilt.

Mit diesen Definitionen lassen sich anspruchsvolle Eigenschaften wie Sicherheits- oder Lebendigkeitsanforderungen über die Laufzeit eines Programms nach folgenden Regeln definieren[BK08].

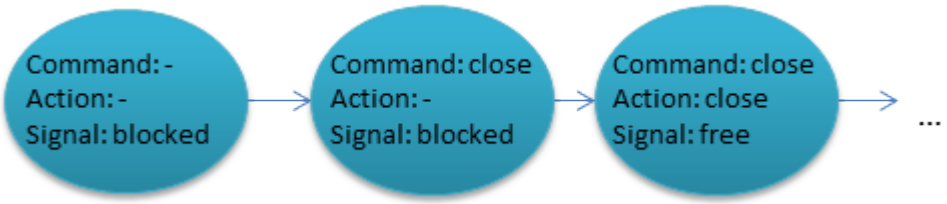


Abbildung 2.3: Visualisierung der LTL Formel *G (¬(Action=close) U (Signal=free))*

$$\lambda ::= true \mid a \mid \lambda_1 \wedge \lambda_2 \mid \neg\lambda \mid X\lambda \mid \lambda_1 U \lambda_2$$

Mit einer so definierten Regel kann eine Ausführungssequenz auf Gültigkeit einer Eigenschaft, mit Bezug auf die Änderung des Zustandspfades über die Laufzeit, betrachtet werden. Dabei ist mit 'Änderung des Zustandspfades' die Betrachtung von Zustandsübergängen gemeint, also die Änderung der Ausführungsattribute. Um die Formulierung von Eigenschaften in LTL Syntax zu demonstrieren stelle man sich eine Zugtür vor, diese darf sich nicht schließen wenn jemand im Weg steht. Nun sei *close* der Befehl des Systems die Tür zu schließen und *free* das Signal das der Einstieg frei ist, dann drückt *G (¬(Action=close) U (Signal=free))* aus dass die Tür sich erst schließen darf wenn der Einstieg frei ist.

Darstellung als Büchi Automat Eine weitere sehr praktische Eigenschaft der LTL ist das verschiedene Algorithmen [VW94][KMMP93] existieren um einen äquivalenten Büchi Automaten zu konstruieren. Dieser bildet den Grundstein für die Verifikation einer Eigenschaft mittels expliziter Model Checking.

NNF Jede LTL Formel kann in negative Normal Form übersetzt werden, was unter anderem für im BMC(siehe Kapitel 2.3.5) benötigt wird, indem Negationen nur direkt vor atomaren Funktionen erlaubt werden [BCC+03]. Negationen in einer LTL Formel können mittels Anwendung der Regeln nach De-Morgan und den Dualitäten zwischen den temporalen Operatoren verschoben werden;
ein Beispiel wäre:

$$\begin{aligned} LTL_0 &= \neg G(pRq) \\ &\equiv F\neg(pRq) \\ &\equiv F\neg(\neg pU\neg q) \end{aligned}$$

Diese Arbeit konzentriert sich vor allem auf die von Abdulkhaleq und Wagner [AW15a] vorgestellte Methode, mittels derer die Ergebnisse einer STPA Analyse direkt in LTL ausgedrückt werden.

2.3.2 Computation Tree Logic

CTL ist die zweite von zwei, hier vorgestellten, temporalen Logiken die im Bereich der Modellprüfung zum Einsatz kommen. Anders als bei der im letzten Kapitel vorgestellten LTL betrachtet die CTL eine Programmausführung nicht als lineare Folge von Zuständen sondern als Baumstruktur d.h. das jeder Systemzustand mehrere mögliche Folgezustände haben kann. Um diese mehrdimensionale Betrachtung zu realisieren beinhaltet die Definition von CTL Formeln zwei Komponenten. Die Überprüfung der Zustandsvariablen geschieht durch Bildung von Zustandsformeln die nach folgenden Regeln aufgebaut sind:

$$\lambda ::= true \mid a \mid \lambda_1 \wedge \lambda_2 \mid \neg\lambda \mid E\varphi \mid A\varphi$$

Wobei A und E genutzt werden um die Gültigkeit auf den nachfolgenden Pfaden wie folgt zu beschreiben:

1. A die Eigenschaft gilt auf **a**llen ausgehenden Pfaden
2. E es *E*xistiert mindestens ein Pfad auf dem die Eigenschaft gilt

Und φ die schon in LTL genutzten temporalen Operatoren X(next) und U(until) nutzt um die, durch A oder E selektierten Pfade, in einem temporalen Kontext zu durchsuchen:

$$\varphi ::= X\lambda \mid \lambda_1 U \lambda_2$$

Auch hier lassen sich wieder die Operatoren *G(globally)* und *F(finally)* wie bei LTL aus den temporal Operatoren kombinieren. Allerdings ändern sich die Dualitäten da die Operatoren immer von den Pfad Quantoren abhängen:

- **G**
  - $[A \mid E]\ \mathbf{G}\ p \equiv [A \mid E](\mathbf{true}\ \mathrm{U}\ p)$
- **F**
  - $[A \mid E]\ \mathbf{F}\ p \equiv \neg[A \mid E]\ \mathbf{G}\ \neg p$

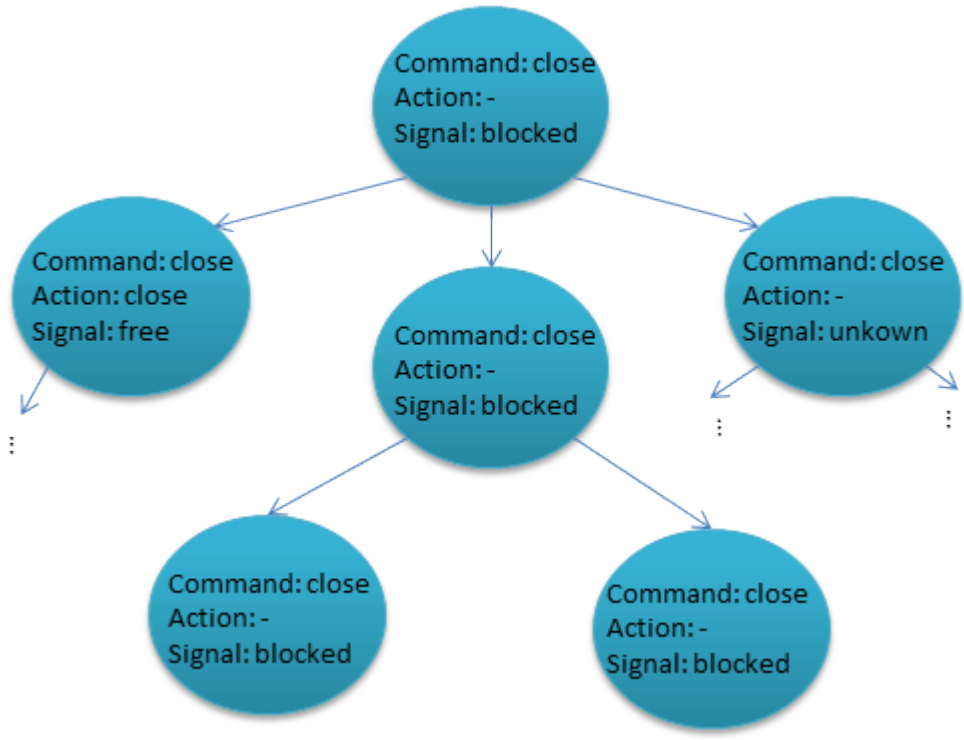


Abbildung 2.4: Visualisierung der CTL Formel AG(¬(Action=close) U (Signal=blocked))

EX p	p gilt in mindestens einem nächsten Zustand auf irgendeinem Pfad
EF p	p gilt in einem Folgezustand auf irgendeinem Pfad
EG p	Es existiert ein Pfad auf dem p dauerhaft gilt
E(p **U** q)	Es gibt einen Pfad auf dem p aktuell oder in einem Folgezustand gilt und auf dem p bis (ein-/ausschließlich) zu diesem Zustand gilt
AX p	p gilt in mindestens einem nächsten Zustand auf irgendeinem Pfad
AF p	p gilt in einem Folgezustand auf irgendeinem Pfad
AG p	Es existiert ein Pfad auf dem p dauerhaft gilt
A(p **U** q)	Es gibt einen Pfad auf dem p aktuell oder in einem Folgezustand gilt und auf dem p bis (ein-/ausschließlich) zu diesem Zustand gilt

Tabelle 2.1: Definitionen der

2.3.3 Spin Model Checker

In diesem Abschnitt soll der von Gerard J. Holzmann [Holo4] Model Checker *Spin* vorgestellt werden. Außerdem wird auf die verschiedenen, in dieser Arbeit referenzierten Funktionen von *Spin* sowie das ebenfalls von Holzmann vorgestellte Werkzeug *Modex*.

Spin ist ein Werkzeug zur Verifikation von Sicherheits- und Lebendigkeitsanforderungen an ein System basierend auf Büchi Automaten [Hol97]. Um die Anforderungen effizient in einem Systemkontext überprüfen zu können müssen diese in Form von LTL Formeln (siehe Kapitel:2.3.1) vorliegen. Diese LTL Formeln können dann von *Spin* „on-the-fly"in Büchi Automaten [Tho90], also formell überprüfbare endliche Automaten, umgewandelt werden. Spin geht davon aus das die Anforderungen als sogenannte „never claims", also Bedingungen die nie auftreten sollen, formuliert sind. Durch diesen Trick ist es möglich die Gültigkeit einer Sicherheitsanforderung zu zeigen indem man beweist das der zu Grunde liegende Automat für keinen erreichbaren Systemzustand terminiert. Falls der Beweis schief geht, zeichnet Spin den akzeptierten Systemzustand auf und speichert ihn als Gegenbeispiel der überprüften Anforderung ab.

Ein so modelliertes System kann dann von *Spin* , in für die betrachtete Problemstellung optimierten C-Code, umgewandelt werden. Dadurch wird das eigentliche Model Checking nicht von *Spin* sondern einem externen „Verifier Programm"durchgeführt.

2.3.4 Promela

Spin ließt Modelle in der Sprache Promela(Process Meta Language), einer prozessorientierten Modellsprache, ein. In Promela kann die Ausführung eines Systems abgebildet und dessen erreichbare Zustände analysiert werden. Um dies zu erreichen bietet Promela die Möglichkeit ein System als Zusammensetzung aus bis zu 256 Prozessen, 255 global oder lokal definierten Kommunikationskanälen sowie beliebig vielen Prozess- oder Systemvariablen zu modellieren.
Ein Prozess p kann als initial aktiv mit *active proctype p()* $\{...\}$
oder als Prozess Definitionen mit *proctype* $p(arg_0, .., arg_n)$ $\{...\}$
modelliert werden wobei im zweiten Fall eine Instanz erst durch *run* $p(arg_0, .., arg_n)$ erzeugt wird. Durch diese Definition ist es möglich ein System entweder gleich mit mehreren aktiven Prozessen zu starten oder mithilfe eines Initiierungsblocks (*Init* $\{...\}$) zu starten.

Prozessinstanzen werden dann asynchron ausgeführt und können untereinander mithilfe global definierter Kommunikationskanäle Nachrichten austauschen. Der Syntax zur Deklaration von Variablen ist analog zum C Syntax, dabei sind die zur Verfügung stehenden Typen:

Typ	Werte	Bsp. Definition	
bool	true,false	**bool** var = true;	
byte	0 ... 255	**byte** var = 2;	
mtype	1 ... 255	mtype = {on, off, error}	*(on = 3, off = 2, error = 1)*
pid	0 ... 255	pid var = run p()	*die Instanz Nummer eines Prozesses*
short	-2^{15} .. $2^{15}-1$	short var = -3	
int	-2^{n} .. $2^{n}-1$	int var = 2	**n ist abhängig von der Prozessorarchitektur**
chan	1 ... 255	chan var = [4] {mtype, int }	*eine Nachricht in var enthält ein mtype und ein int*

Tabelle 2.2: Tabelle der in Promela definierten Variablen Typen [Hol16]

2.3.5 NuSMV Model Checker

In diesem Abschnitt wird der in dieser Arbeit eingesetzte *NuSMV* Model Checker vorgestellt und seine Funktionsweise grob dargestellt. *NuSMV* [McM93] ist wie das im Kapitel 2.3.3 vorgestellte Werkzeug *Spin* ein Model Checker der eingesetzt wird um Zustandsspezifikationen in einem System zu überprüfen. Anders als *Spin* setzt *NuSMV* hier allerdings auf BDD(Binary Decision Diagram) und seit Version 2.0 auch auf ein SAT(satisfaction problem)[CGP+02] basierendes symbolisches Model Checking. Hierbei kann *NuSMV* ein System welches als endlicher Automat, in der *NuSMV* Eingabesprache (siehe Kapitel 2.3.6) vorliegt, gegen Anforderungen in LTL, CTL oder PSL sowie auch gegen Invarianten überprüfen.

Wie auch bei *Spin* ist *NuSMV* rein kommandozeilenbasiert wobei der Nutzer hier entscheiden kann ob er eine Verifikation entweder durch Ausführung eines klassischen Batch-Befehls oder durch Aufruf von NuSMV im interaktiven Modus durchführen will. Beide Modi bieten die Möglichkeit sowohl SAT- als auch BDD-basiertes Model Checking durchzuführen. Da aber die Batch-Methode lediglich im Modell definierte Sicherheitsanforderungen berücksichtigt und bei weitem nicht die Konfigurationen bietet, beschränkt sich dieses Dokument auf den interaktiven Modus. Abhängig von der gewünschten Verifikationsmethode, wobei zu beachten ist das die SAT Methode nur LTL Formeln akzeptiert, sind folgende Schritte durchzuführen:

- beim **BDD Model Checking** konstruiert *NuSMV* ein binäres Entscheidungsdiagramm aus dem Systemmodell. Wenn ein so konstruiertes BDD vorliegt kann entweder eine im Modell oder direkt in der Shell definierte Anforderung in CTL oder LTL überprüft werden. Hierzu sei erwähnt dass das BDD Model Checking wie der Name schon sagt auf Entscheidungen basiert und damit auf die Überprüfung von CTL Anforderung beschränkt ist [CGP+02]. Um BDD basiertes LTL Model Checking durchzuführen reduziert [EMC09] *NuSMV* das Problem auf die Verifikation einer CTL Anforderung durch Konstruktion eines Tableaus(auch Kripke Struktur) welches alle Zustände der Potenzmenge der atomaren Funktionen der LTL Formel beinhaltet.

- beim **BMC(Bounded Model Checking)** wird ein **Erfüllbarkeitsproblem** aus dem vorliegenden Transitionen System M, in Form einer Kripke Struktur [Gum], und einer zu überprüfenden LTL Eigenschaft ϕ, in NNF, gebildet. Der Verifikationsprozess besteht darin für sämtliche Pfade auf die Erfüllung von ϕ hin zu überprüfen. Wobei die Suchtiefe durch einen gegebenen Umfang ('bound') beschränkt wird (siehe [BCC+03].

2.3.6 Die NuSMV Eingabesprache

In diesem Kapitel wird eine generelle Übersicht über die Eingabesprache des NuSMV Model Checkers gegeben wie sie im Benutzerhandbuch zur der Anfang 2016 aktuellen Version 2.0.6 [1]. Dabei ist das Ziel dieses Kapitels nicht den genauen Aufbau eines NuSMV Modells darzustellen; dafür wird auf die Spezifikation der Sprache in der gewünschten Version verwiesen die auf der NuSMV Homepage[2] zu finden ist.
Für den NuSMV Model Checker definierte Modelle von endlichen Zustandsmaschinen bestehen aus einem oder mehreren Modulen. Dabei muss ähnlich wie in C ein „main"-Modul existieren, welches den Einstiegspunkt definiert und jedes weitere Modul als Zustandsvariable enthält. Jedes Modul stellt einen parallelen Prozess dar der im Wesentlichen aus einer Menge von Zustandsvariablen und Zustandsübergangsrestriktionen besteht. Optional kann ein Modul, bis auf das „main"-Modul, noch eine Menge an Aufrufparametern, in Klammern hinter dem Bezeichner, definieren. Dabei beginnt die Definition einer Zustandsvariablen mit dem Codewort *VAR* gefolgt von dem gewählten Variablennamen gefolgt von einer oder mehrerer Variablennamen und einem der Typenbezeichner:

1. **boolean**: für eine boolesche Variable
2. **n .. m**: für eine Integervariable mit dem die Werte aus dem Intervall [n,m] annehmen kann, wobei$n > -2^{31}$ und $m < 2^{31}$
3. $\{n_0, n_1, ..., n_m\}$: In NuSMV kann ein Enum durch Definition einer Folge, deren Glieder Integer- oder String-Konstanten seien können.
4. $[signed, unsigned]\ word[N]$ wobei ein Wort der Länge drei (N=3) ein drei stelliger Bit Vektor zur Speicherung boolescher Werte ist.
5. $modul_name$ ein Modul kann durch Instanziierung der Ausführungssequenz hinzugefügt werden, hierbei müssen alle vom Modul verlangten Parameter „by-value"übergeben werden.

Ein simples Beispiel ist das hier gegebene Programm, welches einen einfachen „Modulo Vier Zähler"definiert:

```
MODULE main
VAR
```

[1] http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf
[2] http://nusmv.fbk.eu/

```
counter : 0 .. 10;
ASSIGN
init(counter) := 0;
next(counter) := (counter + 1) mod 4;
```

Hier wird nur das notwendige „main“-Modul benutzt welches die Zustandsvariable *counter* mit dem Intervall [0,10] definiert. Durch das Schlüsselwort **ASSIGN** wird *counter* erst auf null initialisiert und dann durch *next()* in jedem weiteren Zustand auf *(counter + 1) mod 4* gesetzt.
NuSMV akzeptiert sowohl LTL als auch CTL, *timed* CTL, Invarianten und PSL Spezifikationen die entweder dem Kommandozeilenbefehl oder der *NuSMV* Shell übergeben wurden oder direkt im Modell definiert sind. Aufgrund des Kontextes dieser Arbeit, die sich lediglich auf die Spezifikation von CTL und LTL beschränkt, seien hier kurz die Eigenheiten der Formulierungen aufgezeigt:

- LTL Spezifikationen

G [n,m] p	*bounded gobally*	wahr wenn p in allen Zuständen, die minimal n und maximal m Schritte in der Zukunft liegen, gilt
F [n,m] p	*bounded finally*	wahr wenn p in allen Zuständen, die minimal n und maximal m Schritte in der Zukunft liegen, irgendwann gilt
Y p		wahr wenn p im letzten Zustand galt
Z p		wahr wenn der aktuelle Zustand der initiale ist oder **Y** p gilt
H p	*historically*	wahr wenn p in allen vorausgegangenen Zuständen wahr war (vgl. G)
H [n,m] p	*bounded historically*	wahr wenn p in allen Zuständen, die maximal n und minimal m Schritte zurückliegen, galt
O p	*once*	wahr wenn p mindestens einmal in der Vergangenheit wahr war (vgl. F)
O [n,m] p	*bounded once*	wahr wenn p in allen Zuständen die maximal n und minimal m Schritte zurückliegen mindestens einmal galt
p **S** q	p *since* q	wahr wenn p seit einem Zeitpunkt $t^{'}$, $t^{'}$ nicht eingeschlossen, an dem q galt, gilt
p **T** q	p *triggered* q	wahr wenn p zu einem Zeitpunkt $t^{'}$ galt und q seit einschließlich dieses Zeitpunktes gilt. Wenn p nie galt dann **H** *q*

Tabelle 2.3: Erweiterung der temporalen Operatoren für die Definition von CTL Spezifikationen in NuSMV 2.6

- CTL Spezifikationen

E [p**U** q **]**	*exists*	wahr wenn es einen Pfad gibt so dass,
	until	p auf dem gesamten Pfad gilt bis q wahr wird
A [p**U** q **]**	*forall*	wahr wenn auf allen Pfaden gilt dass,
	until	p auf dem gesamten Pfad gilt bis q wahr wird

Tabelle 2.4: Erweiterung der temporalen Operatoren für die Definition von CTL Spezifikationen in NuSMV 2.6

2.3.7 Model Extraction

Um ein Softwaresystem in ein Modell für einen Model Checker zu übersetzten gibt es mehrere denkbare Möglichkeiten [AW15a]. Eine sehr elegante Lösung bietet das von Bell Laboratories veröffentlichte Werkzeug *Modex* [HHS01]. *Modex* übersetzt C-Code direkt in Promela wodurch man in der Lage ist Systemmodelle direkt auf Implementierungsebene abzuleiten. *Modex* ist lediglich als Quellcode verfügbar und muss lokal kompiliert und installiert werden. *Modex* ist für eine Unix/Linux Umgebung konzipiert, wodurch eine Installation hier lediglich die auf der Homepage des Projektes[3] beschriebenen Schritte benötigt [4].
Die Kompilierung auf Windows Rechnern benötigt eine Cygwin[5] Umgebung. Das innerhalb der Cygwin Umgebung enthaltene Cygwin-Terminal kann *Modex* wie unter Linux mit folgendem Aufruf installiert werden:

- *$ make install*
- *$ ./make_pc*

make_pc ist ein 'make' Script welches für die Kompilierung mittels des Visual Studio C++ Compilers geschrieben wurde,

[3]http://www.spinroot.com/modex/
[4]http://spinroot.com/modex/MANUAL.html
[5]http://www.cygwin.com/

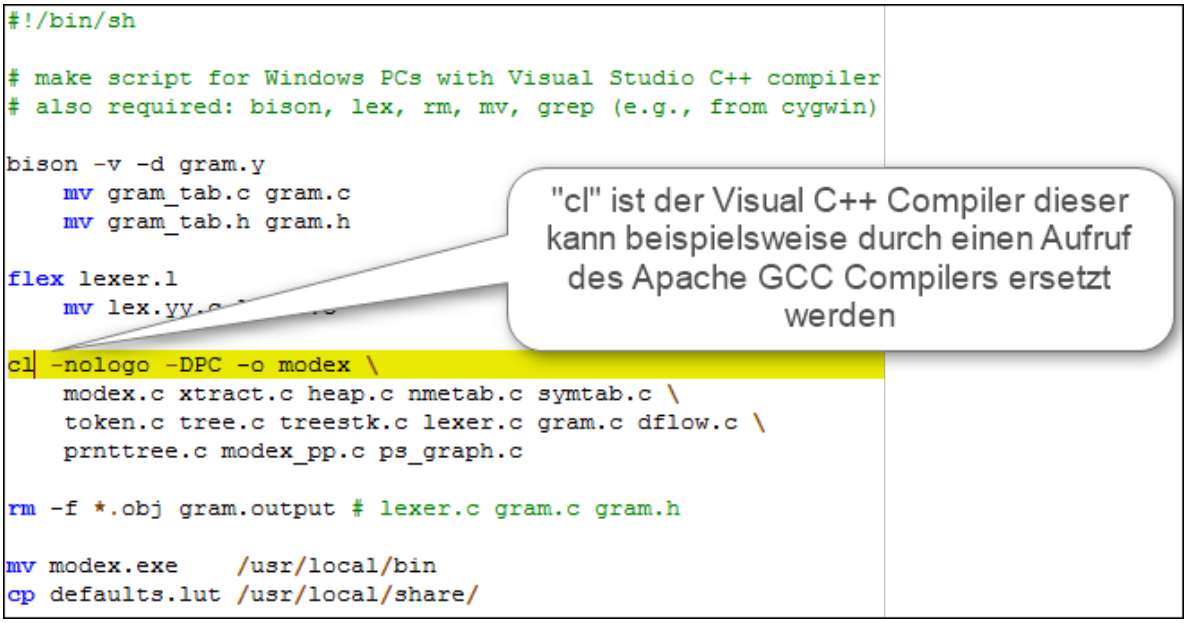
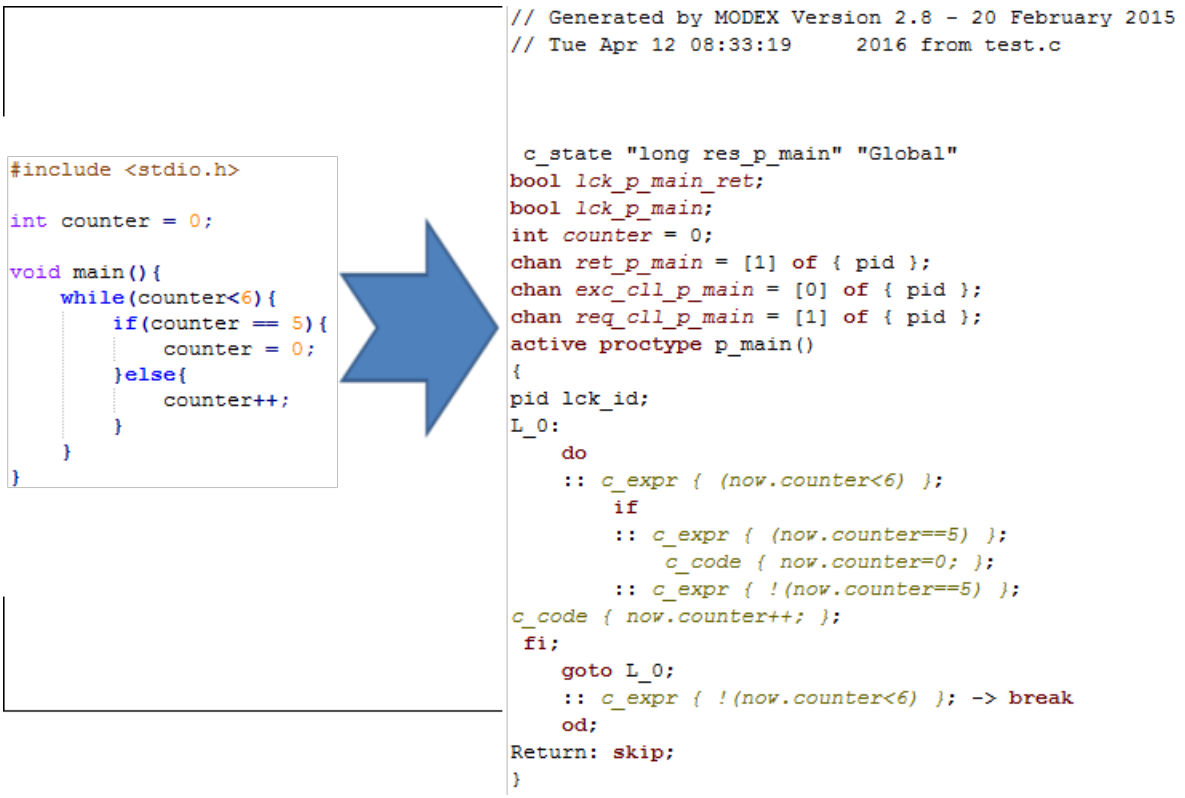


Abbildung 2.5: Das "make_ pcSScript welches in den Modex 2.8 Quelldateien beigelegt ist

Auf diese Weise kann man beliebig komplexe und große Systeme, zu welchen der Quellcode in C vorliegt, in Promela übersetzen. Hier muss man allerdings genau auf die Typenwahl der definierten Zustandsvariablen, also der Variablen die man zur Verifikation benutzen möchte, achten. Aufgrund der deutlich kleineren Auswahl an Typen in Promela die in Tabelle 2.2 dargestellt sind werden nur Variablen vom Typ Integer, Boolean, Short und Byte von *Modex* direkt in das Promela Modell geschrieben. Sämtliche anders definierten Variablen werden mit dem Codewort **c_ state** als eingebetteter C Code hinzugefügt.



```
#include <stdio.h>

int counter = 0;

void main(){
    while(counter<6){
        if(counter == 5){
            counter = 0;
        }else{
            counter++;
        }
    }
}
```

```
// Generated by MODEX Version 2.8 - 20 February 2015
// Tue Apr 12 08:33:19     2016 from test.c


 c_state "long res_p_main" "Global"
bool lck_p_main_ret;
bool lck_p_main;
int counter = 0;
chan ret_p_main = [1] of { pid };
chan exc_cll_p_main = [0] of { pid };
chan req_cll_p_main = [1] of { pid };
active proctype p_main()
{
pid lck_id;
L_0:
    do
    :: c_expr { (now.counter<6) };
        if
        :: c_expr { (now.counter==5) };
            c_code { now.counter=0; };
        :: c_expr { !(now.counter==5) };
c_code { now.counter++; };
 fi;
    goto L_0;
    :: c_expr { !(now.counter<6) }; -> break
    od;
Return: skip;
}
```

Abbildung 2.6: Ein Beispiel für die automatische Extraktion eines Promela Modells aus C-Code durch *Modex* 2.8

2.4 STPA SwISS: STPA for Software-Intensive Systems Approach

Der von Abdulkhaleq et al. [AWL15] vorgeschlagene Ansatz nutzt den in Kapitel 2.2 vorgestellten STPA Prozess um Sicherheitsanforderungen an ein System auf Systemebene abzuleiten und mittels Model Checking zu verifizieren. STPA SwISS nutzt hierfür einen symbolischen oder expliziten Model Checker für die Verifikation auf Implementierungsebene. In einem letzten Schritt werden aus den abgeleiteten Sicherheitsanforderungen Testfälle generiert und ausgeführt.
Im Detail handelt es sich also um einen Prozess welcher die Ausführung folgender drei Schritten erfordert:

1. Ableitung von Sicherheitsanforderungen auf Systemebene durch Anwendung des STPA Prozesses inklusive den vorgeschlagenen Ergänzungen von Thomas [Tho13] und Abdulkhaleq.
2. Erstellung eines „safe behavior models“

3. Dieser Schritt teilt sich in zwei Unteraufgaben wodurch versucht wird die Vorteile von sowohl formaler als auch dynamischer Software Sicherheitsverifikation einzufangen.

   3.1. Die gegebenen Anforderungen müssen zunächst, mithilfe des durch Abdulkhaleq und Wagner in [AW15a] präsentierten Algorithmus zur Ableitung formaler Software Sicherheitsanforderungen, in die in Kapitel 2.3.1 vorgestellte LTL, übersetzt werden. Diese werden dann mithilfe des in Kapitel 2.3 vorgestellten Ansatzes des Model Checkings gegen ein vorher aus dem Code (siehe Kapitel 2.3.7) oder aus dem im letzten Schritt erstellten „safe behavior models“abgeleiteten Verifikationsmodells überprüft.

   3.2. Den Abschluss des Prozesses bildet das Generieren und Ausführen von Testfällen. Es wurden zwei Methoden vorgeschlagen um Testfälle zu erstellen: Entweder direkt aus eventuellen Gegenbeispielen die während des Model Checkings gefunden wurden oder durch den Einsatz von Modell basierten Testwerkzeugen.

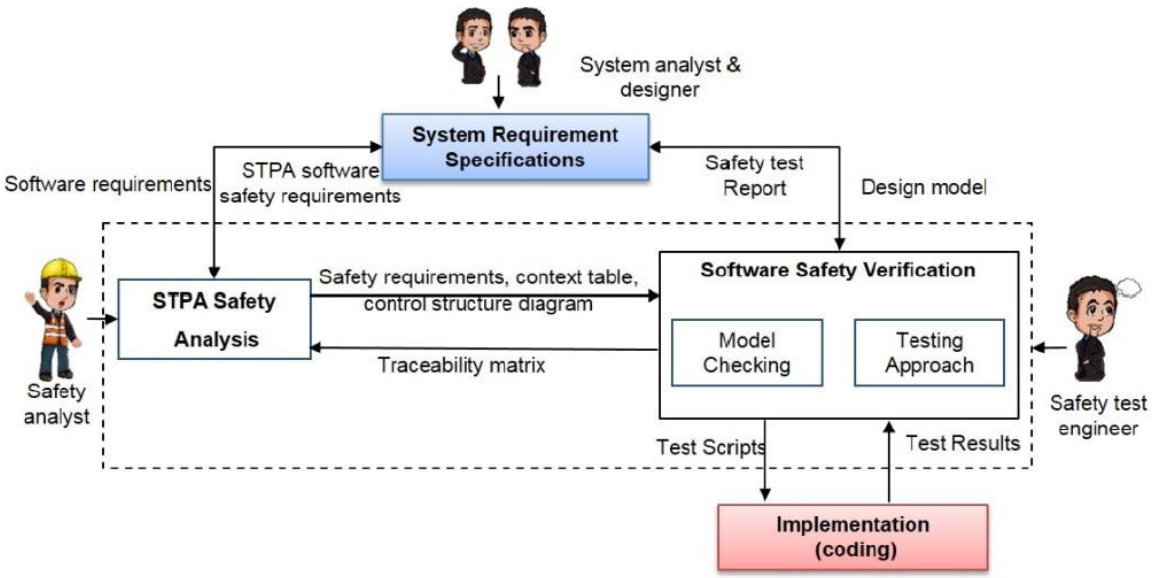


Abbildung 2.7: Ansicht des in [AWL15] vorgeschlagenen Ansatzes zur STPA basierten Sicherheitsverifikation in Software intensiven Systemen

2.5 Tool Unterstützung

An der Universität Stuttgart wurde seit 2013 ein Reihe an Werkzeugen zur der von Leveson vorgestellten Sicherheitstheorie [Lev11] in Form von Softwarelösungen unterstützt. Der aus dieser Arbeit resultierende STPA Verifier soll ebenfalls als Erweiterung dieser Plattform zur Verfügung gestellt werden und als Ergänzung der Funktionalität dienen.

2.5.1 XSTAMPP

XSTAMPP [AW15b] ist eine *Rich Client Platform* die seit Anfang 2015 an der Universität Stuttgart in Java entwickelt und bereitgestellt wird. Basierend auf dem Eclipse RCP Framework bietet *XSTAMPP* eine Basis für die Entwicklung von schritt-basierten Plug-ins sowie Schnittstellen und Grundimplementierungen um die Einbindung dieser Plug-ins einfach zu gestalten. *XSTAMPP* ist aus dem *A-STPA* Werkzeug, welches ebenfalls an der Universität Stuttgart im Rahmen eines Studienprojektes 2013 entwickelt wurde, hervorgegangen. Im Zuge dieser Migration wurde A-STPA als Plug-in, welches in Kapitel 2.5.2 noch genauer vorgestellt wird, der Plattform hinzugefügt. Abbildung 2.8 zeigt eine schematische Darstellung der Architektur von XSTAMPP und der bis zur Veröffentlichung dieser Arbeit veröffentlichten Plug-ins.

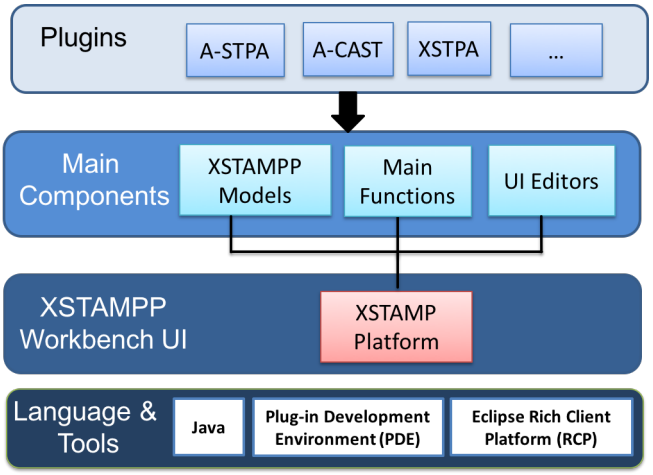


Abbildung 2.8: Die Architektur von XSTAMPP

2.5.2 A-STPA

A-STPA ist ein an der Universität Stuttgart entwickeltes Plug-in für XSTAMPP [AW14a] welches eine Implementierung der in Kapitel 2.2 vorgestellten Gefährdungsanalyse bietet. *A-STPA* Version 2.0.5 stellt dabei die von Leveson [Lev11] vorgeschlagenen Schritte in Form benutzerfreundlicher Editoren, zur Verfügung.

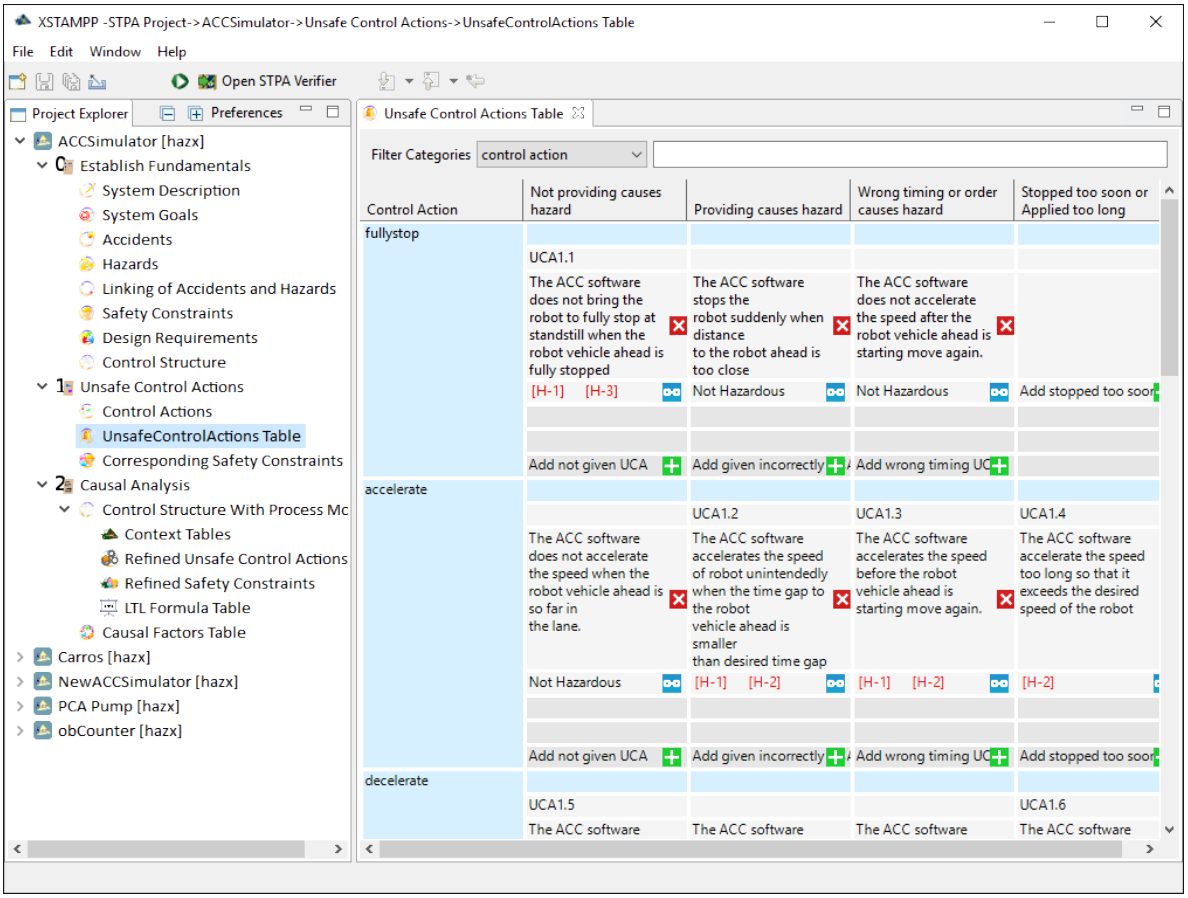


Abbildung 2.9: Die XSTAMPP Plattform 2.0.2 mit einem geöffneten A-STPA 2.0.5 Editor zur Protokollierung unsicherer Regelungsaktionen

2.5.3 XSTPA

XSTPA [AW16] ist eine Ergänzung des A-STPA Plug-ins die dieses um die durch Thomas und Abdulkhaleq vorgeschlagenen Erweiterungen des STPA Prozesses erweitert. XSTPA kombiniert die in [Tho13] vorgeschlagene automatisierte Analyse von unsicheren Regelungsaktionen mit den von Abdulkhaleq und Wagner in [AW15a] aufgestellten Erweiterungen der Kontexttabelle. Diese reduzieren die Größe der Kontexttabelle durch den Einsatz von kombinatorischen Tests. Hierzu nutzt XSTPA den ACTS [Div16] Algorithmus um aus den in den Prozess Modellen definierten Variablen und deren Belegungen Kontexttabellen zu generieren. Aus diesen Tabellen können dann gefährdende Belegungen analysiert und in entsprechende Sicherheitsanforderungen [AW15a] umgewandelt werden.

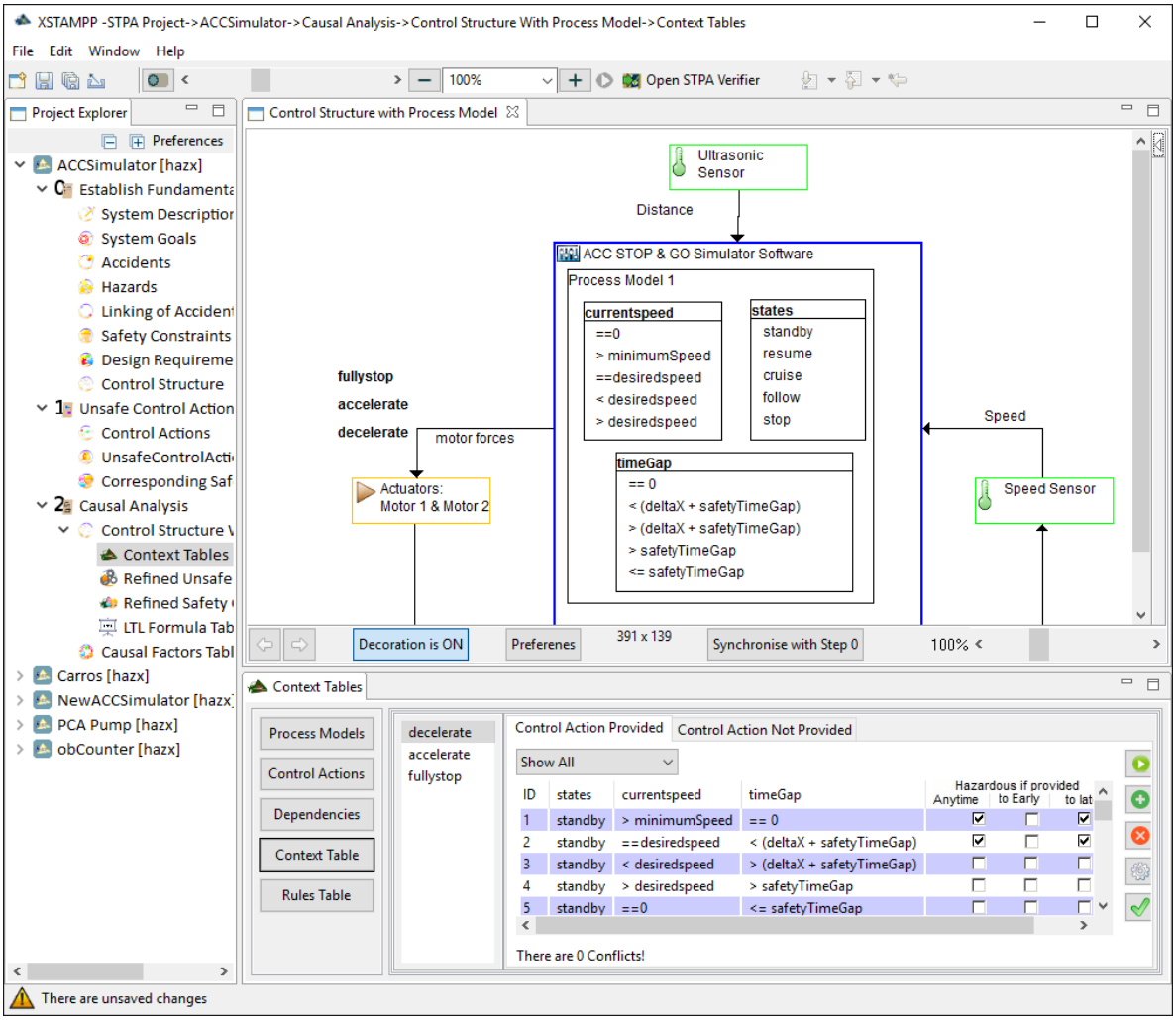


Abbildung 2.10: The XSTAMPP 2.0.2 Platform with XSTPA 1.0.2

3 Analyse und Entwurf

Dieses Kapitel beschäftigt sich mit der Analyse der Problemstellung und der Umsetzung in Form einer konkreten Architektur des STPA Verifier Plug-ins. Ziel dieses Kapitels ist es einen Einblick in die Grundkonzepte der in Kapitel 4 vorgestellten Implementierung zu geben.

3.1 Architektur

In diesem Abschnitt wird die Architektur des STPA Verifier Plug-ins sowie eine Übersicht über die wichtigsten Kommunikationswege zwischen den Hauptkomponenten beschrieben.

Der STPA Verifier ist als MVC Architektur konzipiert, welche auf Grund der erwünschten logischen wie auch visuellen Einbindung in die XSTAMPP Plattform auf dem Eclipse RCP Framework sowie der XSTAMPP Platform selbst aufgebaut ist.

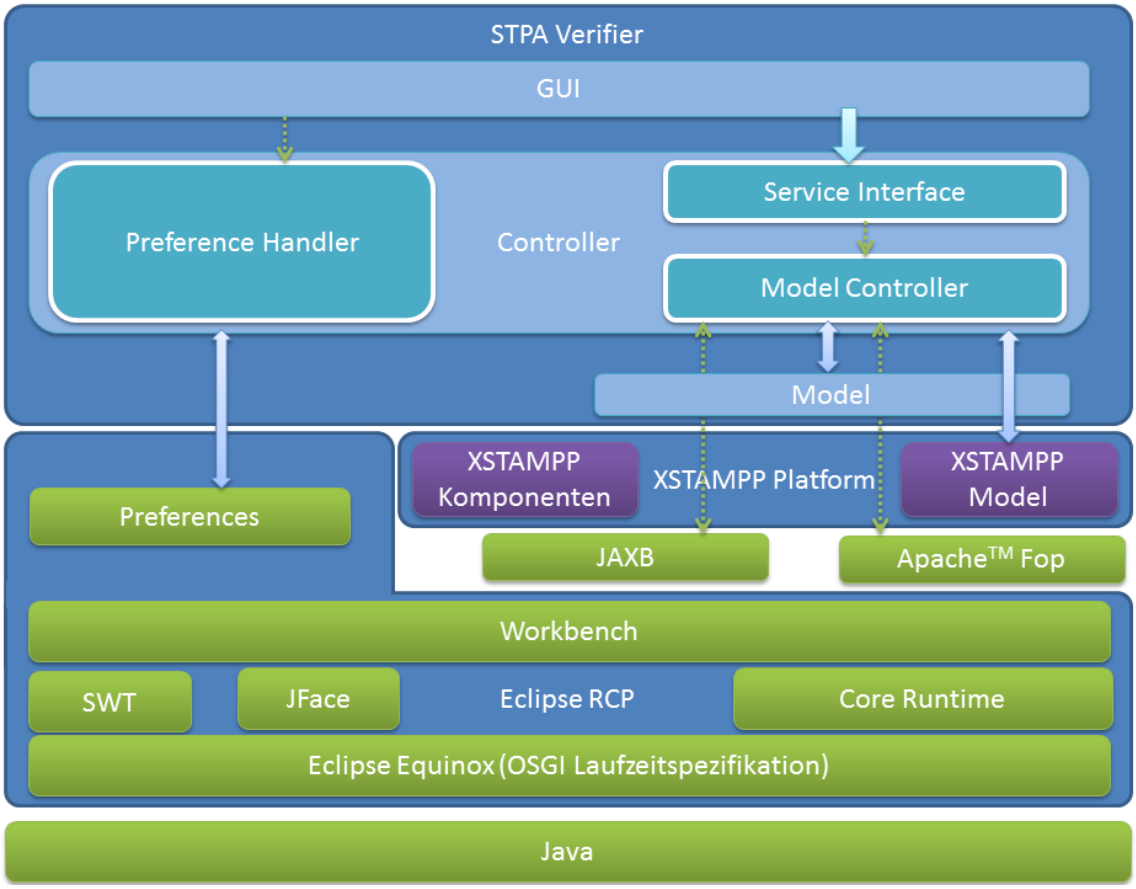


Abbildung 3.1: Architektur des STPA Verifier Plug-in's

Die in Abbildung 3.1 zu sehende zweigliedrige Controller Komponente erlaubt den internen Komponenten eine direkte Kommunikation mit dem Controller, während von extern lediglich eine Serviceschnittstelle erlaubt Anfragen an den Hauptcontroller zu stellen. Des weiteren ermöglicht dieses Design eine klare Trennung zwischen der Verwaltung der Verifikations- und Konfigurationsdaten. Ebenfalls stützt sich der STPA Verifier Export auf die, schon durch XSTAMPP referenzierten, Apache^{TM} FOP[1] und JAXB[2] Bibliotheken.

3.2 Algorithmus

Der STPA Verifier bietet im Wesentlichen folgende drei Grundfunktionen:

[1]https://xmlgraphics.apache.org/fop/
[2]https://jaxb.java.net/

1. Die Eingabe von Sicherheitsanforderungen als CTL oder LTL Spezifikationen bzw. den Import aus einem vorhandenen A-STPA Project (siehe Kapitel 2.5.2)
2. Die Verifikation der eingegebenen Sicherheitsanforderungen mittels des Spin oder NuSMV Model Checkers gegen ein ausgewähltes Systemmodell.
3. Erzeugung eines Verifikations Reports.

Zu diesem Zweck ist der STPA Verifier sowohl Eingabemaske für LTL/CTL Spezifikationen und Konfigurationsdaten als auch graphische Oberfläche zur übersichtlichen Erstellung und Dokumentation von Sicherheitsverifikationen.

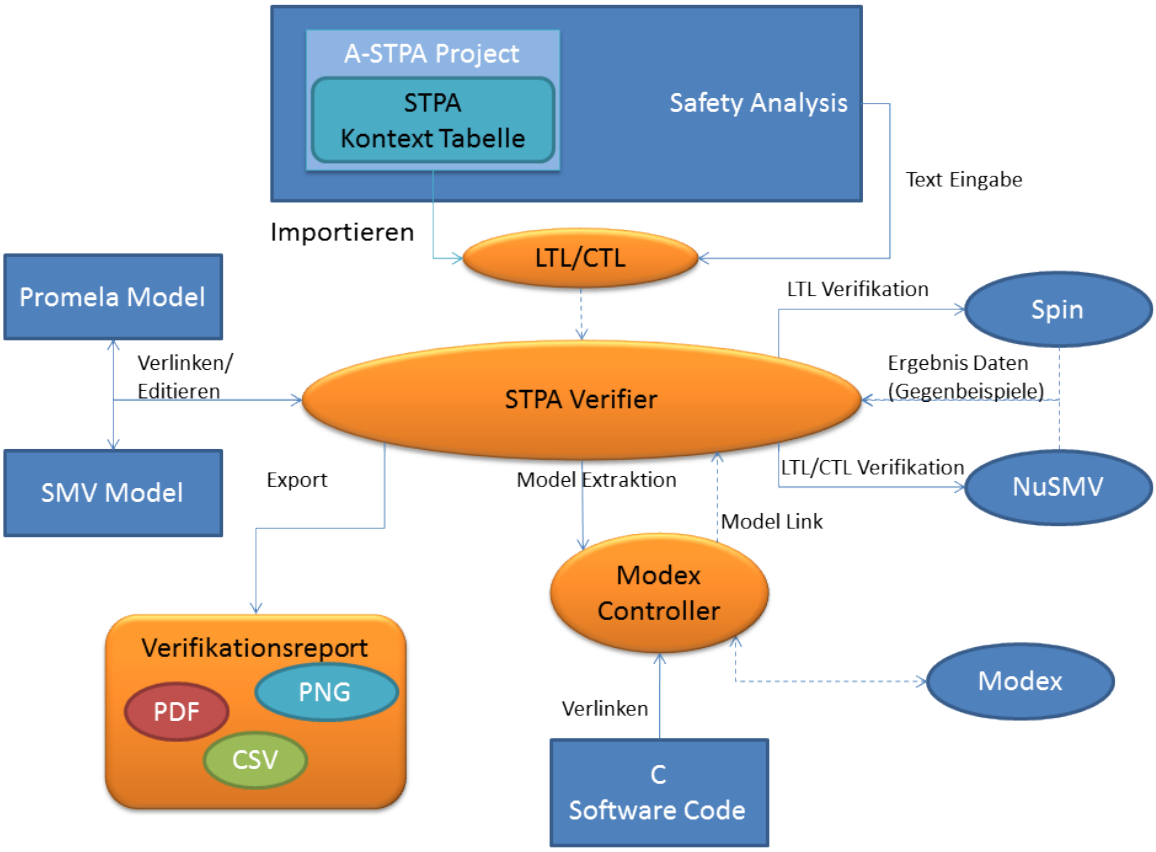


Abbildung 3.2: Algorithmus des STPA Verifier Plug-ins basierend auf dem Konzept von STPA SwISS [AWL15]

Abbildung 3.2 zeigt eine grobe Darstellung des implementierten Algorithmus, wobei die in blau dargestellten Komponenten externe Dateien symbolisieren.

3.3 Klassendiagramme

Das folgende Kapitel soll einen Eindruck über die Paketstruktur und den Aufbau der Programmlogik geben. Die dargestellten Klassendiagramme sind deshalb auf das Wesentlichste beschränkt. Die Klassenhierarchie kann gut in die drei Komponenten des zugrunde liegenden „MVC Pattern“eingeteilt werden. Jedoch wurde hier die klassische Interpretation des Entwurfsmusters, wie in den Abbildungen 3.3 und 3.4 zu sehen, insofern abgewandelt als dass der Controller in zwei Teile aufgespalten ist. Dies hat sowohl praktische als auch designtechnische Gründe, da der STPA Verifier die Daten der Verfikationsläufe nicht über die Laufzeit des Programmes hinweg speichert. Allerdings werden sämtliche Konfigurationseingaben wie Parameter für die Model Checker oder Programmpfade gespeichert.

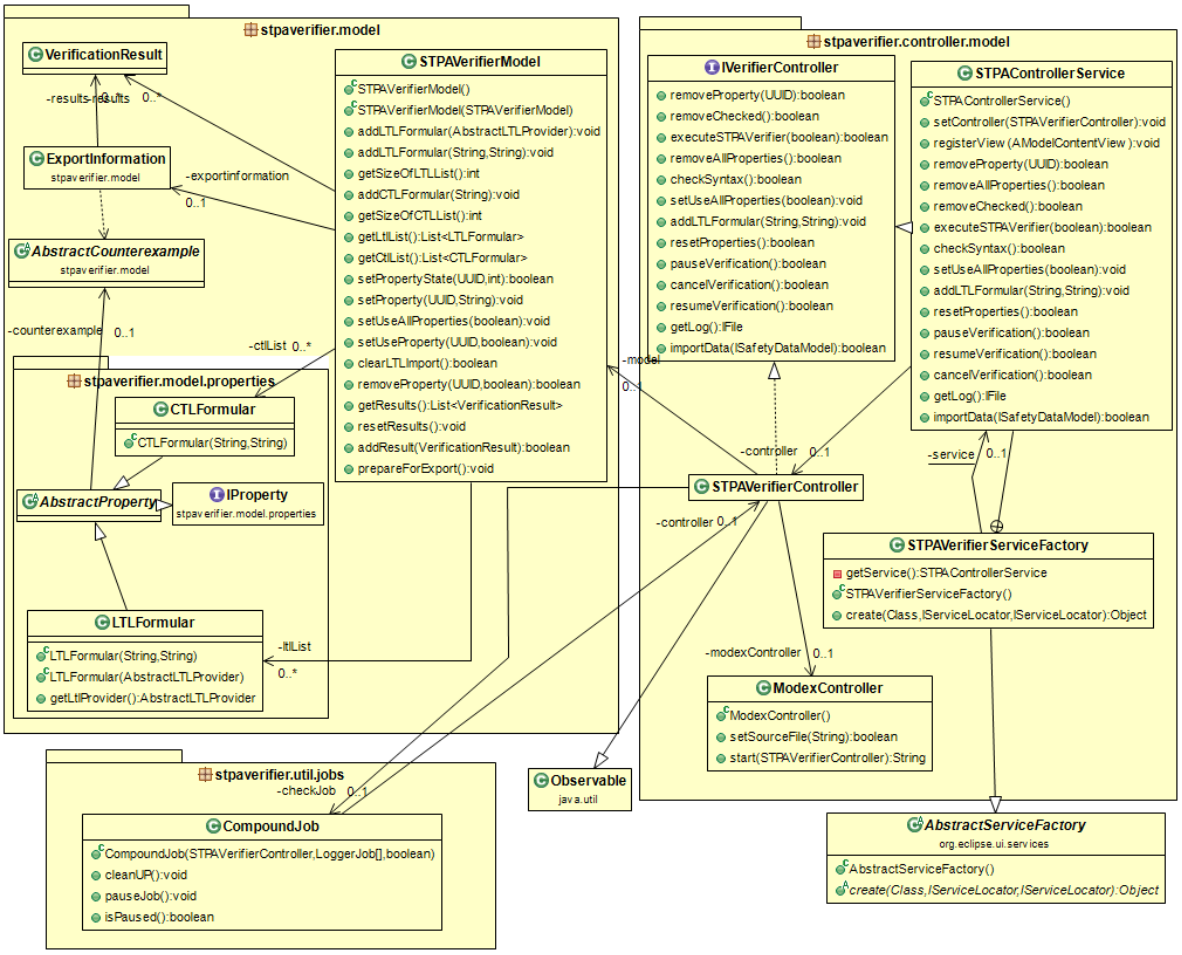


Abbildung 3.3: Klassendiagramm des *stpaVerifier.model* und *stpaVerifier.controller.model* Pakets

Abbildung 3.3 zeigt den für die Ein- und Ausgabedaten der Verifikationen zuständigen Controller. Wie aus dem Diagramm ersichtlich publiziert der *STPAVerifierController* sämtli-

che Änderungen am Daten Modell durch Implementierung des „Observer Patterns“. Die Kommunikation mit dem STPAVerifierController ist über die Erweiterung und Zurverfügungstellung der *org.eclipse.ui.services.AbstractServiceFactory* gelöst wodurch die Klasse indirekt über die *IVerifierController* Schnittstelle angesprochen werden kann.

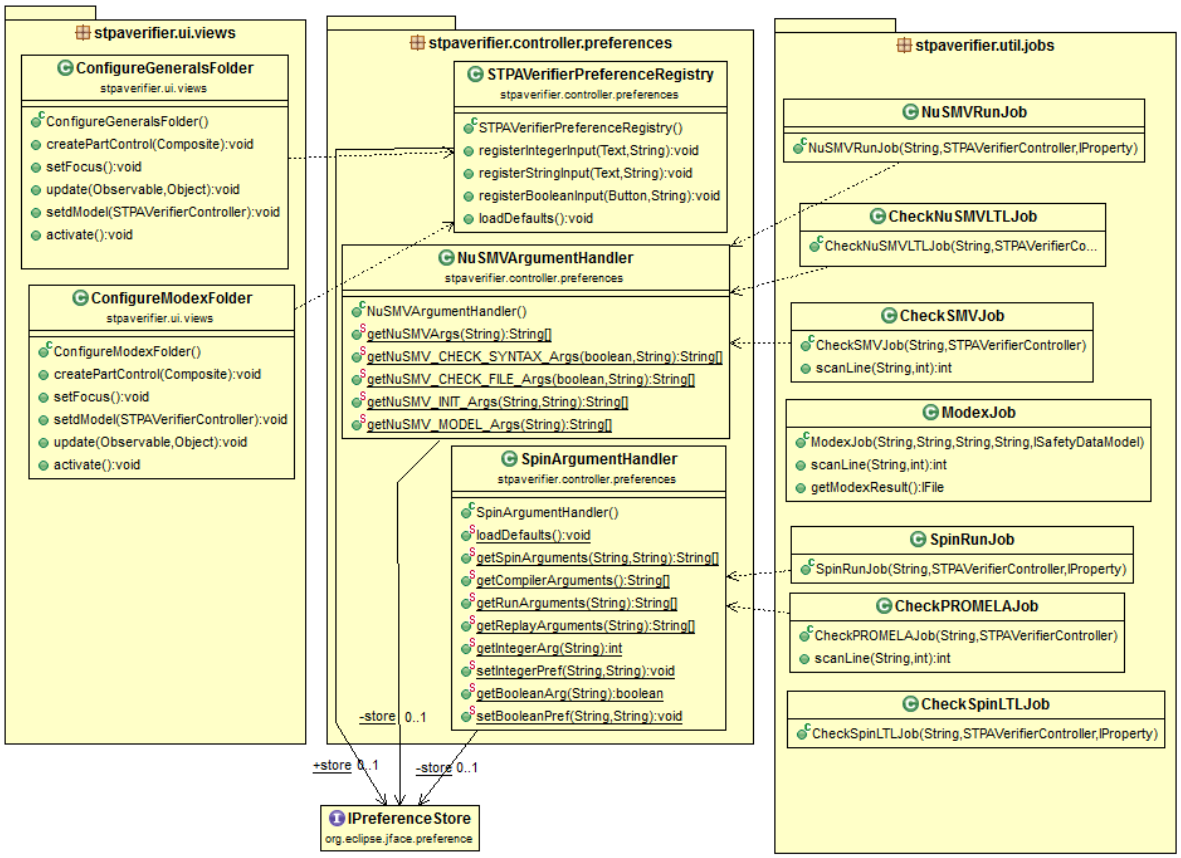


Abbildung 3.4: Darstellung des *stpaVerifier.controller.preferences* Pakets und den Beziehungen mit den UI Konfigurationsklassen und dem *stpaVerifier.util.jobs* Paket

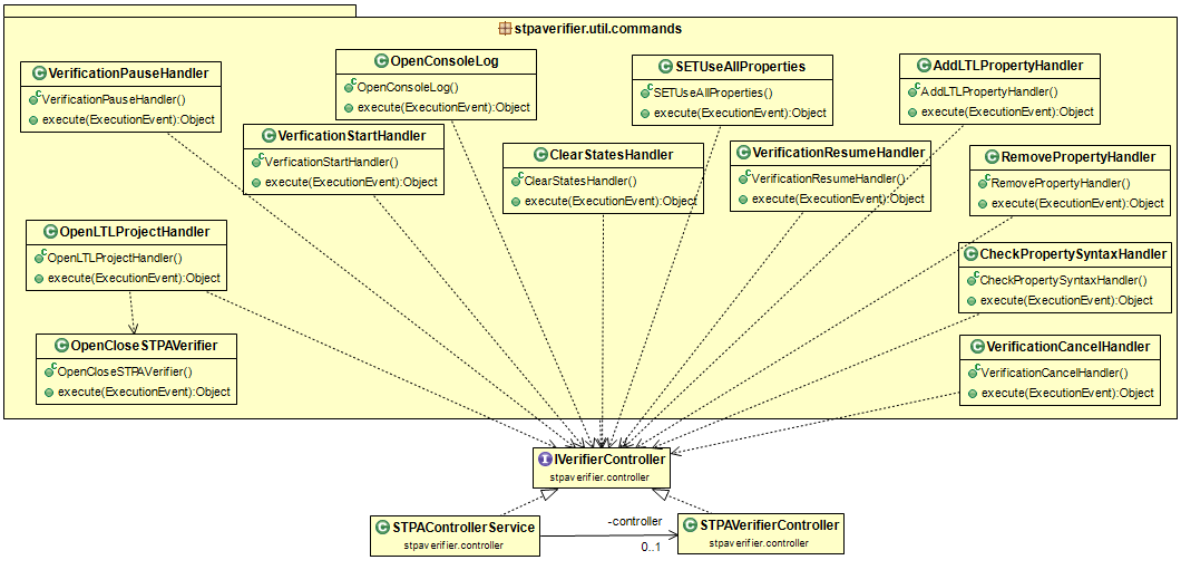


Abbildung 3.5: Klassendiagramm des *stpaVerifier.util.commands* Pakets

3.4 GUI Entwurf

Als Abschluss von Kapitel 3 werden in diesem Abschnitt der Entstehungsprozess und die Konzepte der STPA Verifier Oberfläche sowie deren Umsetzung und Anpassung im Laufe des Projektes vorgestellt.

Die Oberfläche des STPA Verifiers orientiert sich an den Anforderungen einen möglichst übersichtlichen und schnellen Zugriff auf die in Abschnitt 3.2 dargestellten Funktionen zu bieten. Dadurch bedingt wurde im Entwurf bereits eine „Ein-Fenster“Lösung angestrebt. Die Idee hinter diesem Konzept liegt darin alles vor Augen zu haben um die Anzahl an Maus-Klicks um eine Einstellung zu ändern, eine Formel anzupassen oder das Modell einzusehen so gering wie möglich zu halten.

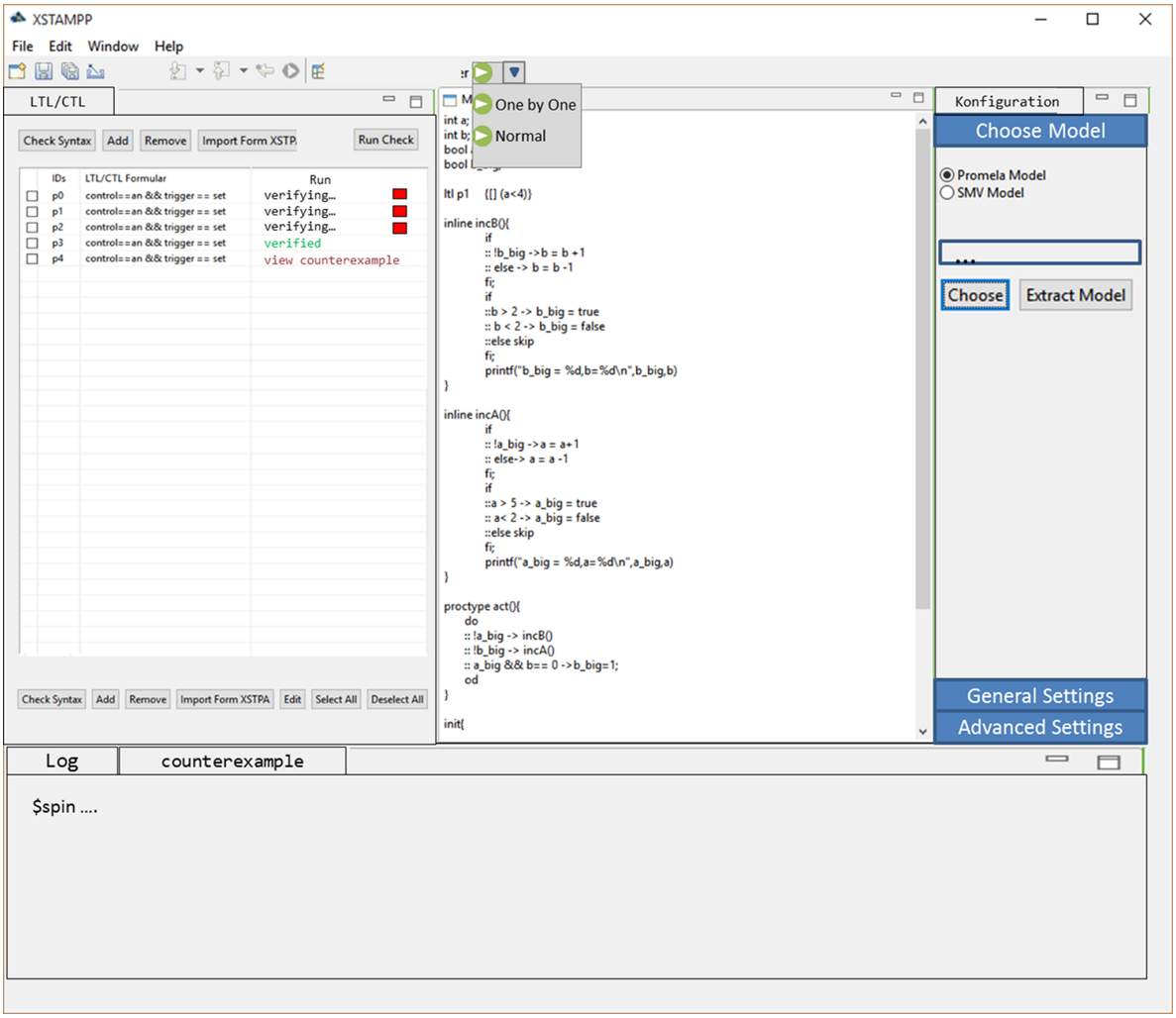


Abbildung 3.6: Der finale GUI Entwurf des STPA Verifiers

Die oben zu sehende Abbildung zeigt den so entstandenen Entwurf der Oberfläche. Wenn man diesen theoretischen Entwurf mit der entstandenen Oberfläche vergleicht fällt vor allem auf das die Integration in die vorhandene XSTAMPP Oberfläche deutlich zugenommen hat. Grund hierfür ist das die direkte Interaktion mit dem A-STPA Projektbaum eine deutlich intuitivere Möglichkeit des Datenimports bietet und durch das Hinzukommen der Öffnen/Schließen-Buttons eine gleichzeitige Nutzung anderer in XSTAMPP integrierter Funktionen möglich ist.

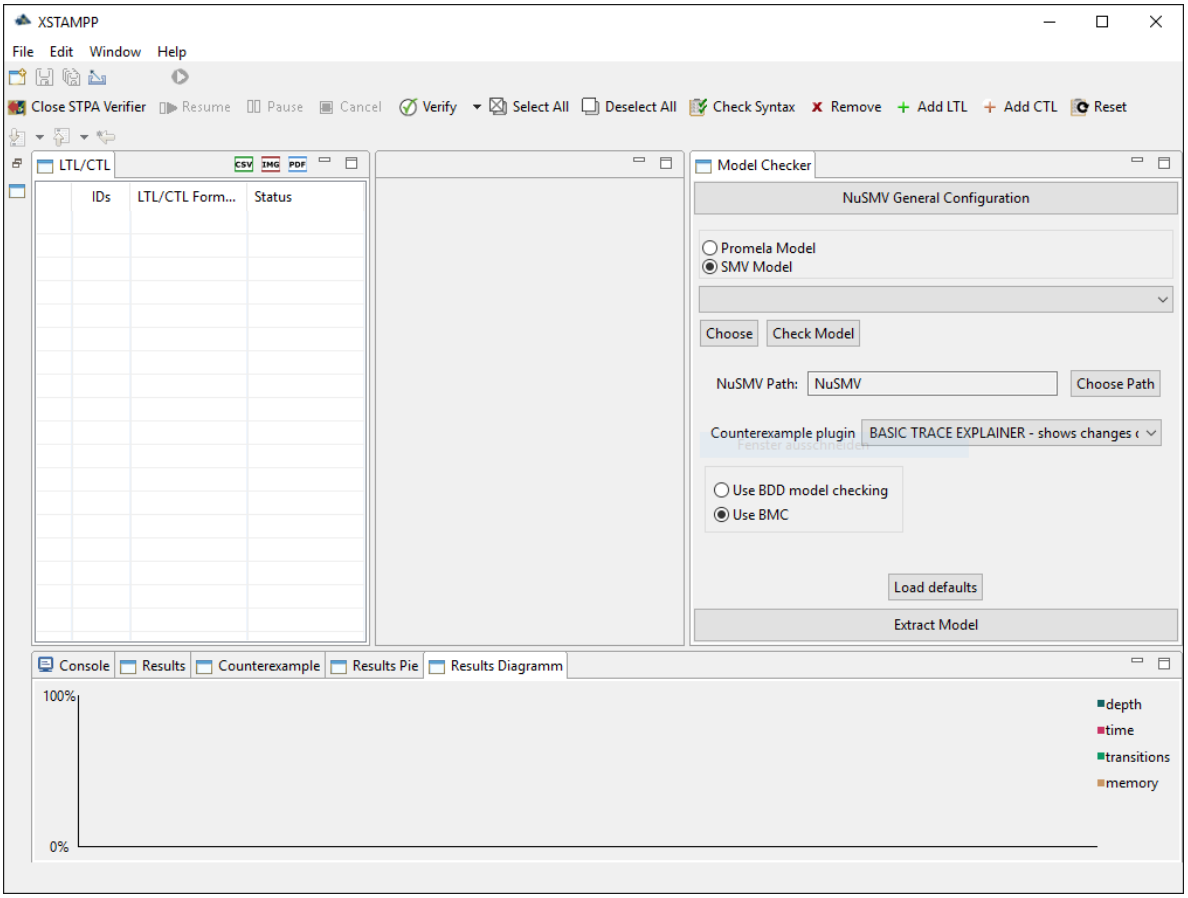


Abbildung 3.7: Oberfläche der STPA Verifiers in der in dieser Arbeit vorgestellten Version 1.0.0

4 Implementierung

4.1 Funktionen

Dieses Kapitel gibt eine theoretische Übersicht über die Funktionen und Fenster des STPA Verifiers, in Version 1.0.0 bieten welche in Kapitel 5 in Form eines praktischen Anwendungsbeispieles vorgeführt werden. Weiter wird ein Eindruck darüber vermittelt werden wie diese eine Software Verifikation unterstützen und damit den, mit herkömmlichen Methoden verbundenen, Aufwand reduzieren können.

4.1.1 LTL Import von A-STPA Projekten

Eines der Hauptziele dieser Arbeit ist auf Grundlage der STPA Gefahrenanalyse, die eine automatisierte Analyse von LTL Sicherheitsanforderungen durch die Erweiterungen von Thomas und Abdulkhaleq ermöglicht, Model Checking zu betreiben. Diese Funktion bietet der STPA Verifier durch eine direkte Kommunikation mit dem A-STPA Plug-in ab Version 2.0.5 über die von XSTAMPP gestellte Schnittstelle *ISafetyDataModel*. So existiert bei gleichzeitiger Ausführung von A-STPA 2.0.5 und dem STPA Verifier ein Menüeintrag *„Import LTL“* wenn man im Projekt Explorer mit der rechten Maustaste auf ein *STPA Projekt* klickt. Diese Funktion importiert alle LTL Formeln, die zuvor in XSTPA ab Version 2.0.2 erstellt werden können, in die dafür vorgesehene Tabelle in der STPA Verifier Perspektive.
Wichtig: Das Entfernen/Editieren von Sicherheitsanforderungen die von einem STPA Projekt importiert wurden hat keinen Einfluss auf das zu Grunde liegenden Projekt

4.1.2 Verwalten der Sicherheitsanforderungen

Abbildung 4.1: Die Toolbar für die Verwaltung der CTL/LTL Tabelle

Die obenstehende Abbildung zeigt die Toolbar die alle Befehle beinhaltet die dem Nutzer bereitgestellt werden um die in Abbildung 4.2 dargestellte Tabelle aller LTL und CTL Formeln zu verwalten.

- **Select All** markiert alle Sicherheitsanforderungen, was durch einen Haken in der Check Box in der ersten Spalte des jeweiligen Tabelleneintrages zu erkennen ist. Eine durch diesen Befehl oder durch manuelle Markierung der Check Box gekennzeichnete Sicherheitsanforderung wird in Verifikationssequenzen, Syntaxüberprüfungen und dem Entfernen Befehl berücksichtigt bzw. mit eingebunden.
- **Deselect All** entfernt alle Haken in den Check Boxen der Tabelle bzw. entfernt die Markierungen aller Sicherheitsanforderungen.
- **Check Syntax** startet eine Syntaxüberprüfung aller markierten Sicherheitsanforderungen. Die Art der Syntaxüberprüfung ist abhängig davon ob im Konfigurationsmenü Spin oder NuSMV aktiv geschaltet wurde.
- **Remove** entfernt alle markierten Sicherheitsanforderungen aus dem STPA Verifier.
- **Add LTL** fügt einen neuen Eintrag mit einer automatisch generierten Formel ID hinzu. Die so hinzugefügte Sicherheitsanforderung muss in LTL definiert werden.
- **Add CTL** fügt einen neuen Eintrag mit einer automatisch generierten Formel ID hinzu. Die so hinzugefügte Sicherheitsanforderung muss in CTL definiert werden.
- **Reset** setzt den Verifikationsstatus aller Sicherheitsanforderungen zurück auf „unchecked“und entfernt alle Gegenbeispiele, bisherige Verifikationsergebnisse und leert die Verifier Konsole.

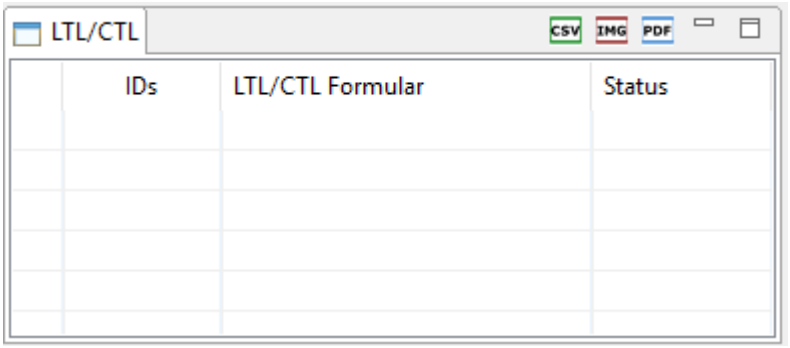


Abbildung 4.2: Die LTL/CTL Formel Tabelle auf der STPA Verifier Oberfläche

4.1.3 Einrichten und konfigurieren der Model Checker

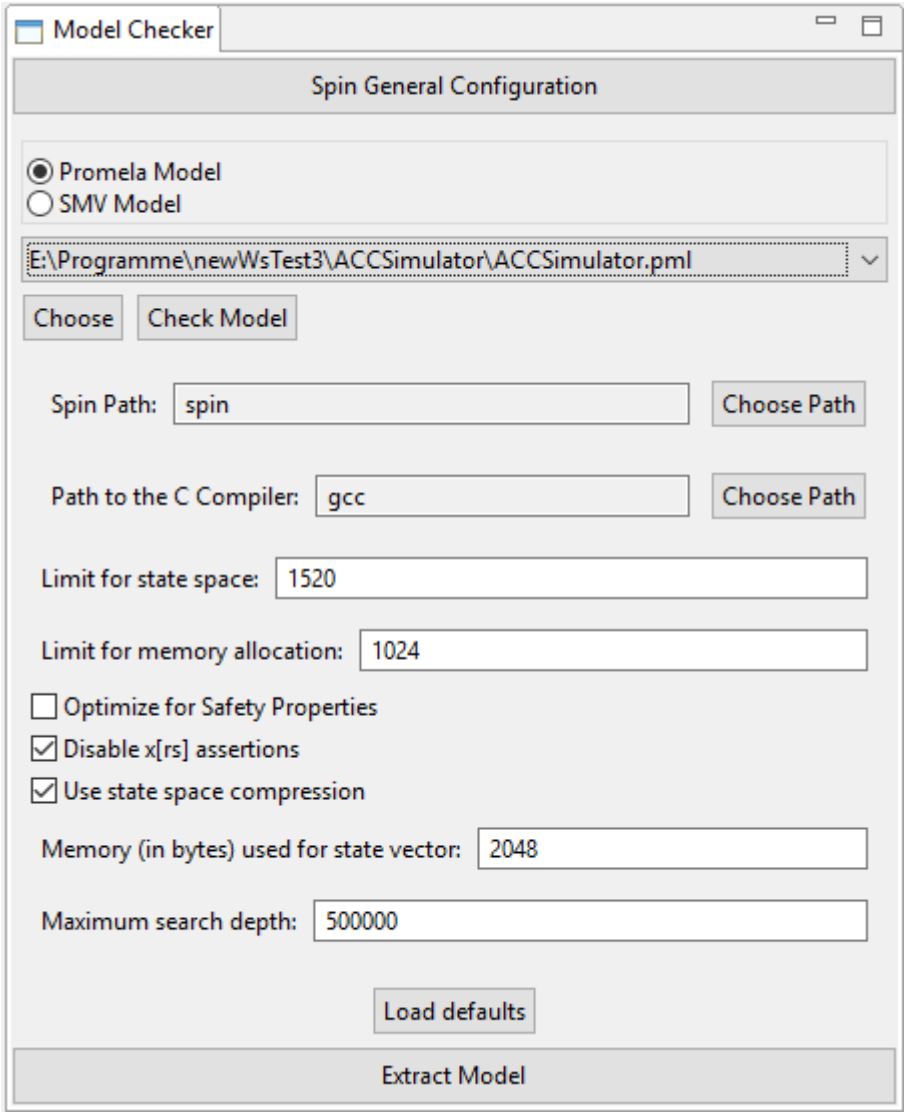


Abbildung 4.3: Die Konfigurationsoberfläche des STPA Verifier in Version 1.0.0

Der Kern des STPA Verifiers ist die Konfiguration und Verlinkung der lokalen Kopien der Model Checker und des zu überprüfenden System Modells. Diese Einstellungen können in dem in Abbildung 4.3 gezeigten View vorgenommen werden. Die so eingegebenen Konfigurationsdaten sind die einzigen in Version 1.0.0 abgespeicherten Daten da diese, anders als die anderen im STPA Verifier vorhandenen Daten, mit dem vom Eclipse RCP Framework zur Verfügung gestellten „preferences“Paket verwaltet werden. So wird erreicht das jedwede Einstellung die vom Benutzer vorgenommen wird dauerhaft im momentan benutzten „Eclipse Workspace“abgespeichert wird. Die einzige Ausnahme von dieser Speicherung bildet die

Angabe der Modelldatei bei der eine solche Speicherung aufgrund der *1..n* Beziehung zwischen Workspace und Modell keinen Sinn ergibt. Stattdessen öffnet die Wahl einer passenden Modell Datei(.smv/.pml) diese in einem Promela oder SMV Editor, der eine Bearbeitung und Abspeicherung des Modells erlaubt. Zwischen Platform und Modell Editor existiert eine *1..1* Beziehung was heißt das maximal ein Modell Editor geöffnet ist der immer das aktuell gewählte Modell anzeigt.

4.1.4 Promela Modell aus C-Code extrahieren

Liegt zu einem Softwarecontroller der Quellcode in ANSI C vor so kann dieser wie in Kapitel 2.3.7 beschrieben mit Hilfe von *Modex* direkt in ein Promela Modell übersetzt werden. Um eine möglichst schnelle und einfache Nutzung dieses Tools direkt von der Oberfläche des STPA Verifiers zu ermöglichen ist die Ausführung von *Modex* direkt über den 'Extract Model' Abschnitt im Konfigurationsmenü möglich. Damit das Plug-in auf Modex zugreifen kann muss die Datei wie in Kapitel 2.3.7 beschrieben auf dem Computer installiert sein und der Link zu der ausführbaren Datei im Eingabefeld „Modex Path“eingetragen oder Modex dem System über die PATH Umgebungsvariable oder eine Installation unter Linux/Mac OS bekannt sein.

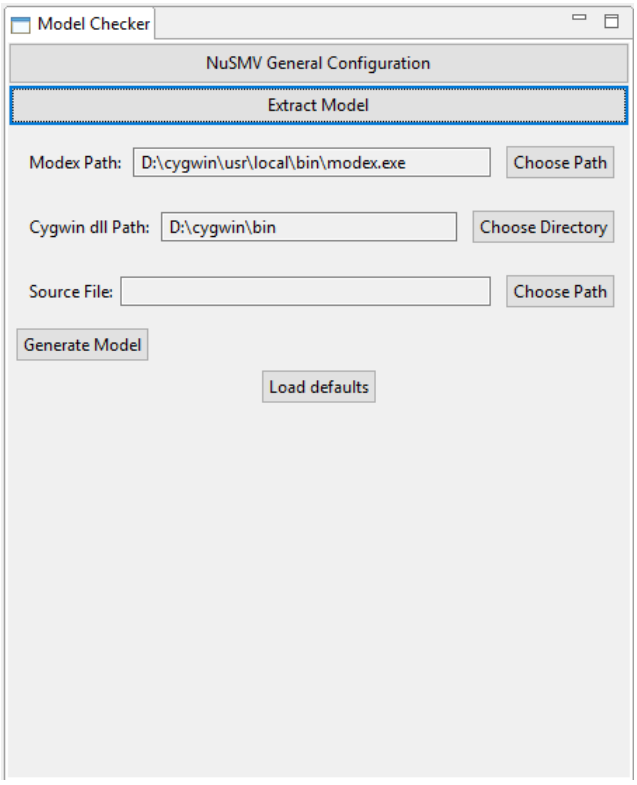


Abbildung 4.4: Die Konfigurationsoberfläche zur Verlinkung und Ausführung von Modex unter Windows(links) und unter Linux (rechts)

4.1.5 Ausführung einer Verifikation

Die Hauptfunktion des STPA Verifiers liegt in der Erstellung und Verifikation von Sicherheitsanforderungen. Während Kapitel 5 die praktische Durchführung einer Verifikation präsentiert, konzentriert sich dieser Abschnitt vor allem auf die internen Prozesse. Um eine Verifikation aller markierten (siehe.: *Verwalten der Sicherheitsanforderungen*) Sicherheitsanforderungen zu starten kann der Benutzer zwischen zwei Optionen wählen:

1. *One by One(Standard)*: Jede einzelne Verifikation einer Sicherheitsanforderung kann während der Laufzeit abgebrochen werden wobei die Ausführungssequenz bei der nächsten Sicherheitsanforderung fortgesetzt wird.
2. *Normal*: Durch Abbruch einer einzelnen Sicherheitsverifikation wird der gesamte Ausführungssequenz gestoppt

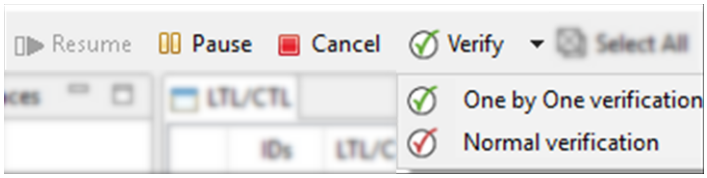


Abbildung 4.5: Die Toolbar für die Ausführung und Kontrolle einer Sicherheitsverifikation während einer Sicherheitsverifikation

Nach Wahl des gewünschten Ausführungstyps erstellt das Programm eine individuelle Ausführungssequenz die immer mit einer Verifikation des Modells beginnt und dann einen Aufruf des gewählten Model Checkers für jede markierte Sicherheitsanforderung anhängt. Während der gesamten Verifikation wird der aktuelle Status der einzelnen Sicherheitsanforderungen in der LTL/CTL Tabelle nach den in Abbildung 4.6 definierten Regeln aktualisiert.

Nr.	Zustand	Zustandsbeschreibung	Zustandsdarstellung in der LTL/CTL Tabelle
0	Ungeprüft	Der Initialzustand	unchecked
1	Syntax korrekt	Die Formel ist durch den momentan aktive Model Checker als syntaktisch korrekt befunden worden	✓ syntax correct
2	Syntax Fehler	Der aktive Model Checker bei der Syntax Überprüfung einen Fehler gefunden, Fehlerdetails werden bei Positionierung des Mauszeigers über der Zustandsanzeige angezeigt	syntax error!
3	Warte auf Ausführung	Eine Verifikation der Sicherheitsanforderung ist geplant, in diesem Zustand kann die Formel nicht Editiert werden	waiting
4	Verifikation wird durchgeführt	Momentan läuft eine Verifikation der Gültigkeit der Anforderung in dem ausgewählten System Modell	processing...
5	Verifikation wurde pausiert	Der geplante Verifikationslauf wurde an dieser Sicherheitsanforderung, durch Betätigung von Pause ,pausiert.	paused
6	Die Verifikation wurde abgebrochen	Temporärer Zustand der den Nutzer über den Abbruch informiert, es gibt drei verschiedene Möglichkeiten in diesen Zustand zu gelangen: 1.: Durch selektieren der Zustandsanzeige während der Ausführung 2.: Durch drücken Cancel in der Toolbar des STPA Verifiers 3.: Durch abbrechen der Verifikation über den Eclipse „Process View“	canceled
7	Verifiziert	Die Gültigkeit der Sicherheitsanforderung wurde durch den Model Checker verifiziert	validated
8	Gegenbeispiel gefunden	Der Model Checker hat bei der Verifikation einen Gegenbeispiel für die Sicherheitsanforderung gefunden, welches durch Selektierung der Zustandsanzeige angezeigt werden kann	failed with Counterexample

Tabelle 4.1: Tabelle der Zustände die eine Anforderung annehmen kann, und deren Repräsentation in der LTL/CTL Tabelle

Um eine solche Ausführungssequenz so effektiv wie möglich gestalten zu können besitzt das Menü zum Starten der Verifikation auch drei Knöpfe die eine Manipulation der Ausführung erlauben:

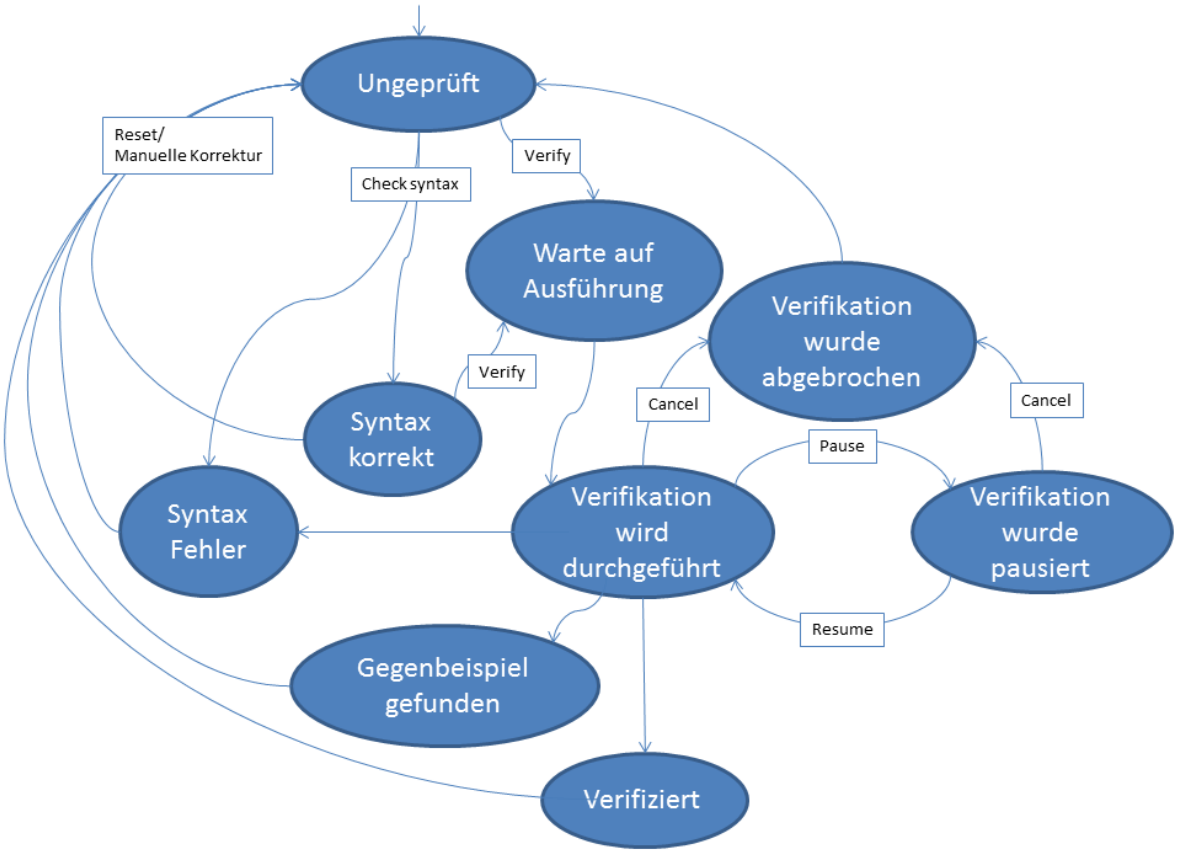


Abbildung 4.6: Diagramm der Zustandsübergänge der Sicherheitsanforderungen

- *Cancel* bricht die Ausführungssequenz unabhängig vom Typ der Ausführung sofort ab (die gleiche Funktion wie die Verifikation einer einzelnen Anforderung im Modus „Normal“abzubrechen)
- *Pause* pausiert die Ausführung und bricht die momentane Verifikation im Bedarfsfall ab.
- *Resume* setzt eine pausierte Ausführung mit der Verifikation, der pausierten Anforderung, fort.

Diese drei Aktionen sind abhängig vom Status der Ausführung und werden, um Fehlbedienung zu vermeiden, ausgegraut wenn sie sie nicht zur Verfügung stehen.

4.1.6 Logging

Der STPA Verifier verfügt über eine eigene Konsole die sämtliche Ausgaben der externen Programmaufrufe in einer Eclipse typischen Konsole anzeigt. Zusätzlich zu der Darstellung

der Informationen zur Laufzeit schreibt der STPA Verifier den Log in eine Log-Datei die in einem pro Modell Datei erstellten Projekt abgelegt ist.

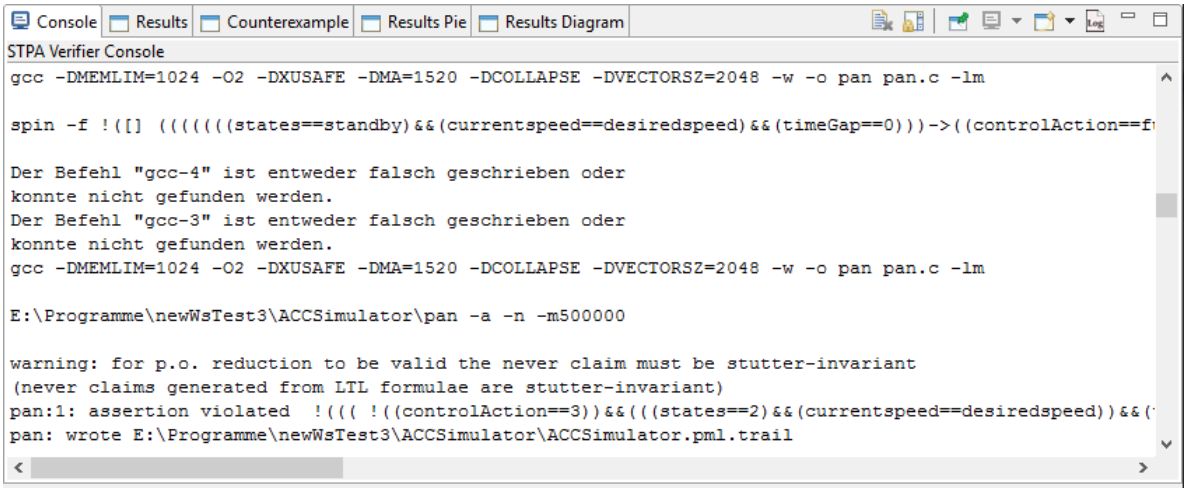


Abbildung 4.7: Die STPA Verifier Konsole zeigt dem Benutzer sämtliche Ausgaben der internen Programmaufrufe

4.1.7 Darstellung der Ergebnisse einer Verifikation

Eine Verifikation von einer oder mehrerer Eigenschaften wird im STPA Verifier durch mehrere Ansichten abhängig vom benutzten Model Checker visualisiert. Hierfür stehen dem Benutzer fünf Komponenten zur Verfügung:

- Die bereits vorgestellte **Statusanzeige**
- Die **Results Table** implementiert die von Abdulkhaleq und Wagner [AW15a] vorgeschlagene Darstellung der Verifikationsergebnisse.
- Der **Counterexample View** bietet eine Darstellung eines gefundenen Gegenbeispiels. Die Art der Darstellung variiert zwischen Spin und NuSMV bzw. den verschiedenen Counterexample Plug-ins die in NuSMV enthalten sind. Das zu benutzende Plug-in kann über die Konfigurationsoberfläche ausgewählt werden.

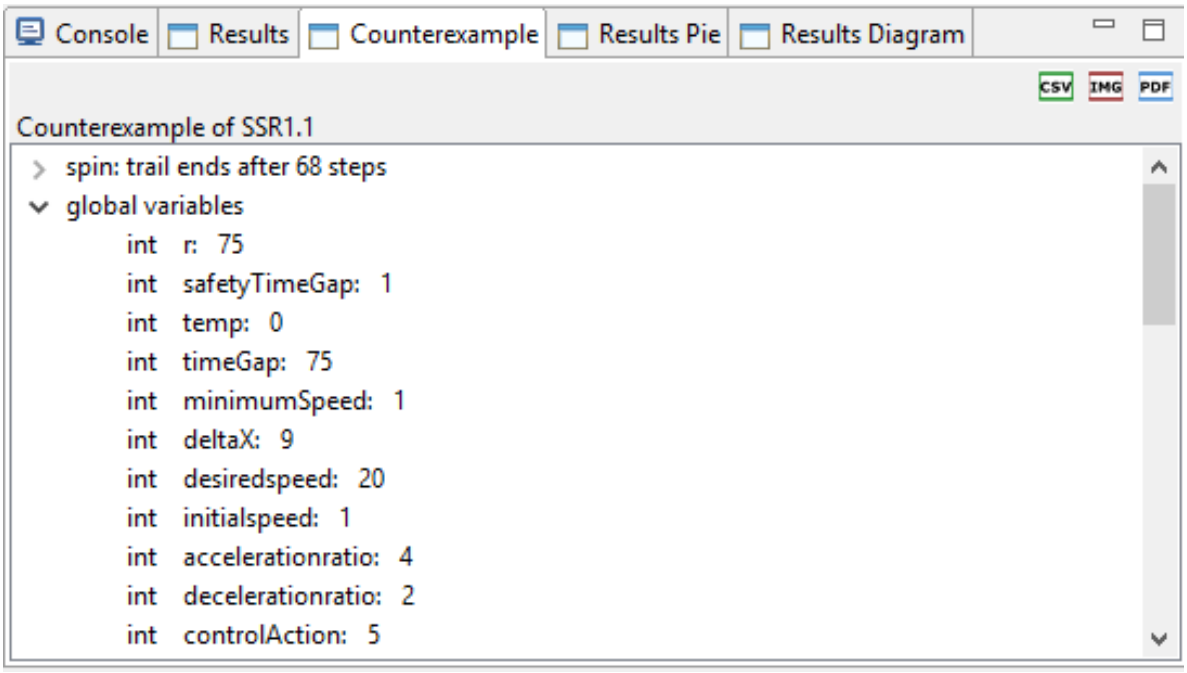


Abbildung 4.8: Darstellung eines Gegenbeispieles für die Anforderung *G(counter<4)* an einen Modulo 6 Zähler in der Counterexample UI des STPA Verifiers

- Der **Results Pie** stellt das Ergebnis einer Verifikation als Kuchendiagramm dar.
- Das **Results Diagram** stellt die Entwicklung der Suchtiefe, der gebrauchten Zeit, den durchlaufenen Transitionen und des genutzten Speicherplatzes auf einer prozentualen Skala über die über der Zeit dar. Dabei ist ein Zeitschritt auf der Y-Achse eine abgeschlossene Verifikation.

Abbildung 4.9: Darstellung des Resultats Diagramms für das in Kapitel 4.1 benutzte Beispiel eines Modulo 6 Zählers

4.1.8 Export der Ergebnisse

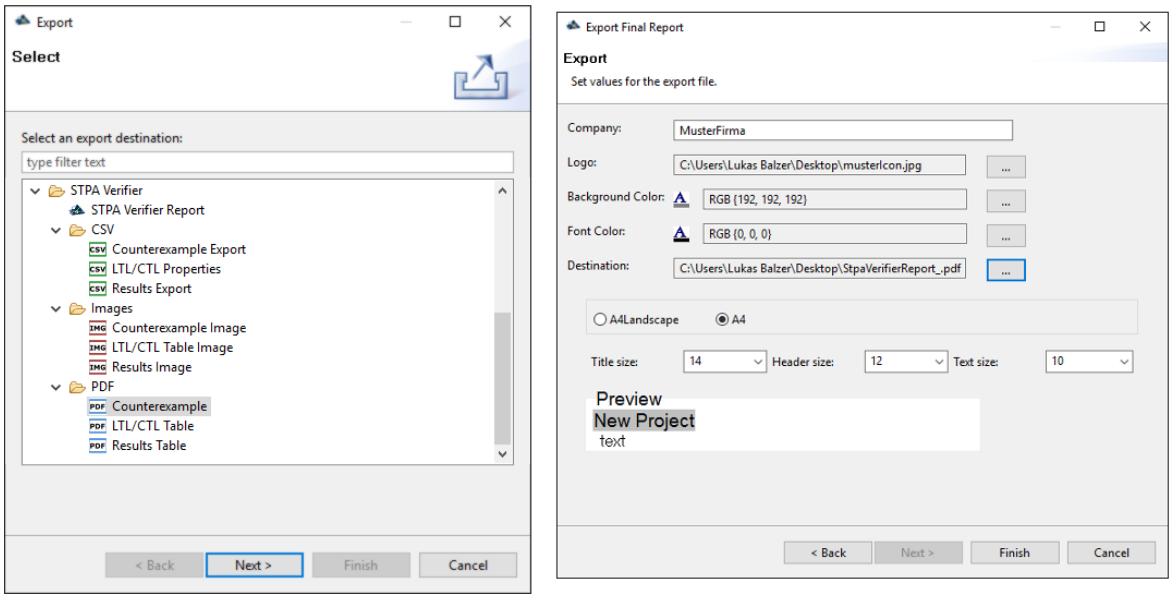


Abbildung 4.10: Die Struktur der STPA Verifier Export Funktionen (links) und Beispielhaft der Wizard eines Verifikations Reports

4.2 Systemtest

In diesem Abschnitt wird der Systemtest für die Zustandsübergangsfunktion und der Kommunikation mit den Model Checkern vorgestellt. Für die Tests wurde das schon im Verlauf dieses Dokuments benutzte Modell einer Modulo 6 Zählerschleife benutzt. Das NuSMV Modell wurde mittels, der vorhandenen Modulo Funktion mit der das Programm in einer Endlosschleife zählt, in der NuSMV Eingabesprache geschrieben. Das Promela Modell wurde aus einem C-Programm, wie schon in Kapitel 2 Abschnitt 2.3.7 als Beispiel angeführt, abgeleitet. Die Funktion und die Zustandsänderungen sind in beiden Modellen trivial.
Für den Test wurden die Formeln in Tabelle 4.2 in die LTL/CTL Tabelle eingetragen und nacheinander durch Spin 6.4.5 und dann durch NuSMV 2.6.0 in den jeweiligen Modellen verifiziert. Die Übersetzung der Formeln in den jeweiligen Syntax wurde automatisch durch die Programm Logik vorgenommen:

- $[] <> (counter == 5) \Rightarrow GF(counter = 5)$
- $[] <> (counter == 6) \Rightarrow GF(counter = 6)$
- $true \Rightarrow TRUE$
- $false \Rightarrow FALSE$

```
MODULE main

VAR
 counter : 0 .. 10;

ASSIGN
 init(counter) := 0;
 next(counter) := (counter + 1) mod 6;
```

```
// Generated by MODEX Version 2.8 - 20 February 2015
// Tue Apr 12 08:33:19     2016 from test.c



 c_state "long res_p_main" "Global"
bool lck_p_main_ret;
bool lck_p_main;
int counter = 0;
chan ret_p_main = [1] of { pid };
chan exc_cll_p_main = [0] of { pid };
chan req_cll_p_main = [1] of { pid };
active proctype p_main()
{
pid lck_id;
L_0:
    do
    :: c_expr { (now.counter<6) };
        if
        :: c_expr { (now.counter==5) };
            c_code { now.counter=0; };
        :: c_expr { !(now.counter==5) };
c_code { now.counter++; };
 fi;
    goto L_0;
    :: c_expr { !(now.counter<6) }; -> break
    od;
Return: skip;
}
```

Abbildung 4.11: Die für den Systemtest verwendeten Modelle in der NuSMV 2.6.0 Eingabesprache(links) und in Promela(rechts)

LTL Formel	**Erwartetes Ergebnis**	**Ergebnis**
$[] <> (counter == 5)$	Verifikation ist erfolgreich	"validated"
$[] <> (counter == 6)$	Es existiert ein Gegenbeispiel	"failed with Counterexample"
$true$	muss per Definition wahr sein	"validated"
$false$	muss per Definition falsch sein	"failed with Counterexample"
$Falsch$	Formel wird nicht erkannt	„syntax error“
	leere Formel produziert einen Syntax Fehler	„syntax error“
$[] <> (zaehler < 6)$	Syntax Fehler da zaehler nicht definiert ist	„syntax error“

Tabelle 4.2: Tabelle der LTL Tests mit erwartetem und angezeigtem Ergebnis der Verifikation

- $Falsche \Rightarrow Falsch$
- $\Rightarrow$
- $[] <> (zaehler < 6) \Rightarrow GF(zaehler < 6)$

5 Anwendungsbeispiel

Um die Funktionen des STPA Verifiers zu testen wurde 2015 an der Universität Stuttgart ein ACC(„Automatic Cruise Control“) mit Start/Stop Simulator[1,2] durch Dennis Maseluk und Asim Abdulkhaleq entwickelt. Hierfür wurde ein Lego MINDSTORM EV3 Roboter mit einer, in ANSI-C erstellten, Simulationssoftware ausgestattet. Das ACC System wurde dann mittels eines EV3 Ultrasonic Sensors in einer Simulation einer Fahrtsituation hinter einem zweiten EV3 Roboter getestet.

Es wurde eine STPA Gefahrenanalyse des Simulators durch Asim Abdulkhaleq mittels der in Kapitel 2.5.1 vorgestellten XSTAMPP Platform durchgeführt, wodurch die Grundlagen einer Sicherheitsverifikation auf Basis des in Kapitel 2.4 vorgestellten Prozesses aufgestellt wurden. Abbildung 5.1 zeigt die für den ACCSimulator aufgestellte Sicherheitsregelstruktur mit allen für die Sicherheitsverifikation benötigten Systemvariablen, die als Basis der Ableitung von formalen Sicherheitsanforderungen benötigt wird.

[1] http://www.iste.uni-stuttgart.de/en/se/forschung/werkzeuge/acc-simulator.html
[2] https://sourceforge.net/projects/acc-with-stop-and-go-simulator/

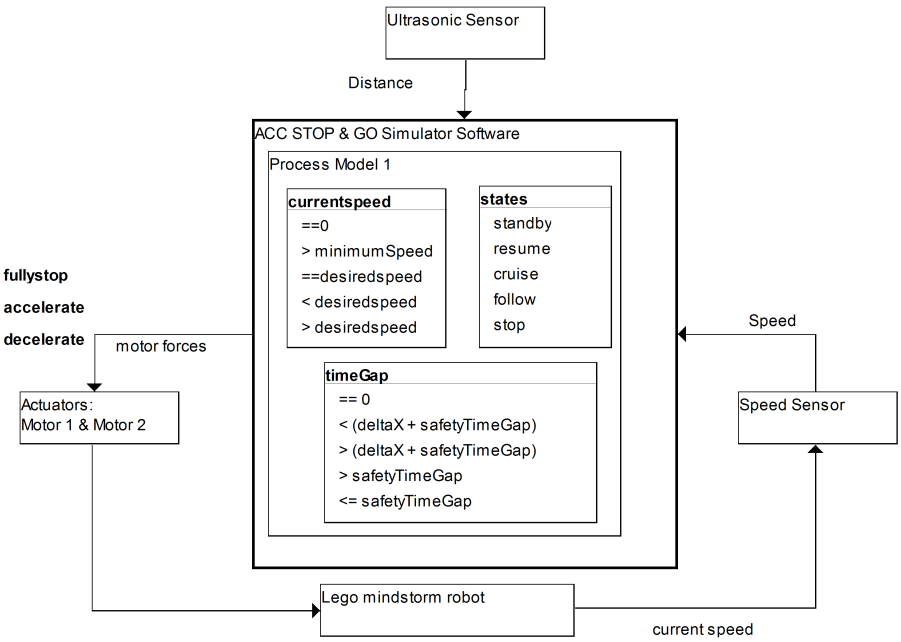


Abbildung 5.1: Sicherheitsregelstruktur, des in ACC stop& go Systems, mit Prozess Modell aus dem sich die LTL Formeln ableiten lassen

In einem zweiten Schritt wurden aus der oben gezeigten Sicherheitsregelstruktur verfeinerte Sicherheitsanforderungen erstellt. Hierzu kam die von Thomas[Tho13] vorgestellte systematische Ableitung von gefährdenden Regelungsaktionen mit den von Asim Abdulkhaleq vorgeschlagenen Verfeinerung der Ergebnisse zum Einsatz.

Refined Safety Constraints Table

ID	Refined Unsafe Control Actions	ID	Refined Safety Constraints
RSR1.1	The fullystop command is provided too late when states is standby and currentspeed is desiredspeed and timeGap is 0	SC2.1	fullystop command must not be provided too late when states is standby and currentspeed is desiredspeed and timeGap is 0
RSR1.2	The fullystop command is provided too late when states is resume and currentspeed is less than desiredspeed and timeGap is 0	SC2.2	fullystop command must not be provided too late when states is resume and currentspeed is less than desiredspeed and timeGap is 0
RSR1.3	The fullystop command is provided too late when states is cruise and currentspeed is greater than desiredspeed and timeGap is 0	SC2.3	fullystop command must not be provided too late when states is cruise and currentspeed is greater than desiredspeed and timeGap is 0
RSR1.4	The fullystop command is provided too late when states is follow and currentspeed is greater than minimumSpeed and timeGap is 0	SC2.4	fullystop command must not be provided too late when states is follow and currentspeed is greater than minimumSpeed and timeGap is 0
RSR1.5	The fullystop command is provided too late when states is stop and currentspeed is desiredspeed and timeGap is 0	SC2.5	fullystop command must not be provided too late when states is stop and currentspeed is desiredspeed and timeGap is 0
RSR1.6	The fullystop command is not provided when states is standby and currentspeed is greater than minimumSpeed and timeGap is 0	SC2.6	fullystop command must be provided when states is standby and currentspeed is greater than minimumSpeed and timeGap is 0
RSR1.7	The fullystop command is not provided when states is standby and currentspeed is desiredspeed and timeGap is less than (deltaX + safetyTimeGap)	SC2.7	fullystop command must be provided when states is standby and currentspeed is desiredspeed and timeGap is less than (deltaX + safetyTimeGap)
RSR1.8	The fullystop command is not provided when states is resume and currentspeed is less than desiredspeed and timeGap is less than (deltaX + safetyTimeGap)	SC2.8	fullystop command must be provided when states is resume and currentspeed is less than desiredspeed and timeGap is less than (deltaX + safetyTimeGap)
RSR1.9	The fullystop command is not provided when states is cruise and currentspeed is less than desiredspeed and timeGap is 0	SC2.9	fullystop command must be provided when states is cruise and currentspeed is less than desiredspeed and timeGap is 0

Abbildung 5.2: Ausschnitt aus der, aus XSTAMPP exportierten, Liste von verfeinerten Sicherheitsanforderungen, die mithilfe des XSTPA Plug-ins Version 1.0.2 und A-STPA 2.0.5 erstellt wurden

XSTPA 1.0.2 leitet außerdem aus den analysierten Variablenbelegungen automatisch formelle Sicherheitsspezifikationen in LTL ab. Dieser Schritt bildet, innerhalb des in Kapitel 2.4 vorgestellten Prozesses, den Übergang von der Gefährdungsanalyse hin zur Sicherheitsverifikation.

LTL Formulas Table

ID	LTL Formulas
SR1.1	[] (((((((states==standby)&&(currentspeed==desiredspeed)&&(timeGap==0)))->((controlAction==fullystop))))&&(! (((((states==standby)&&(currentspeed==desiredspeed)&&(timeGap==0)))U((controlAction==fullystop)))))))
SR1.2	[] (((((((states==resume)&&(currentspeed<desiredspeed)&&(timeGap==0)))->((controlAction==fullystop))))&&(! (((((states==resume)&&(currentspeed<desiredspeed)&&(timeGap==0)))U((controlAction==fullystop)))))))
SR1.3	[] (((((((states==cruise)&&(currentspeed>desiredspeed)&&(timeGap==0)))->((controlAction==fullystop))))&&(! (((((states==cruise)&&(currentspeed>desiredspeed)&&(timeGap==0)))U((controlAction==fullystop)))))))
SR1.4	[] (((((((states==follow)&&(currentspeed>minimumSpeed)&&(timeGap==0)))->((controlAction==fullystop))))&&(! (((((states==follow)&&(currentspeed>minimumSpeed)&&(timeGap==0)))U((controlAction==fullystop)))))))
SR1.5	[] (((((((states==stop)&&(currentspeed==desiredspeed)&&(timeGap==0)))->((controlAction==fullystop))))&&(! (((((states==stop)&&(currentspeed==desiredspeed)&&(timeGap==0)))U((controlAction==fullystop)))))))
SR1.6	[] (((((states==standby)&&(currentspeed>minimumSpeed)&&(timeGap==0)))->((controlAction==fullystop))))
SR1.7	[] (((((states==standby)&&(currentspeed==desiredspeed)&&(timeGap<(deltaX + safetyTimeGap))))->((controlAction==fullystop))))
SR1.8	[] (((((states==resume)&&(currentspeed<desiredspeed)&&(timeGap<(deltaX + safetyTimeGap))))->((controlAction==fullystop))))
SR1.9	[] (((((states==cruise)&&(currentspeed<desiredspeed)&&(timeGap==0)))->((controlAction==fullystop))))

Abbildung 5.3: Ausschnitt aus der, aus XSTAMPP exportierten, Liste von LTL Sicherheitsanforderungen, die mithilfe des XSTPA Plug-ins Version 1.0.2 und A-STPA 2.0.5 erstellt wurden

Zur Durchführung der Sicherheitsverifikation mittels Model Checking wurde ein System Modell für den NuSMV Model Checker auf Basis der Sicherheitsregelstruktur, mit dem von Asim Abdulkhaleq entwickelten STPASTGenerator[3] erstellt. Ferner wurde auch ein Promela Modell mittels *Modex* aus dem ANSI-C Quellcode der Simulationssoftware abgeleitet. Für die Ableitung des Promela Modells wurden die in den STPA Verifier eingebauten Funktion zur Nutzung von *Modex* benutzt. In Kapitel 2.3.3 und 2.3.5 wurden mit Spin und NuSMV zwei häufig eingesetzte Model Checker vorgestellt die, durch den STPA Verifier, in einer automatisierten Verifikation der abgeleiteten STPA Sicherheitsanforderungen gegen die jeweiligen Modelle eingesetzt wurden. Hierzu wurden die in dem oben beschriebenen Prozess entstandenen LTL Spezifikationen, durch den Befehl 'Import LTL' im Kontext Menü des ACCSimulator Projektbaumes im Projekt Explorer der XSTAMPP Oberfläche, in den STPA Verifier übertragen.

[3]https://sourceforge.net/projects/stpastgenerator/

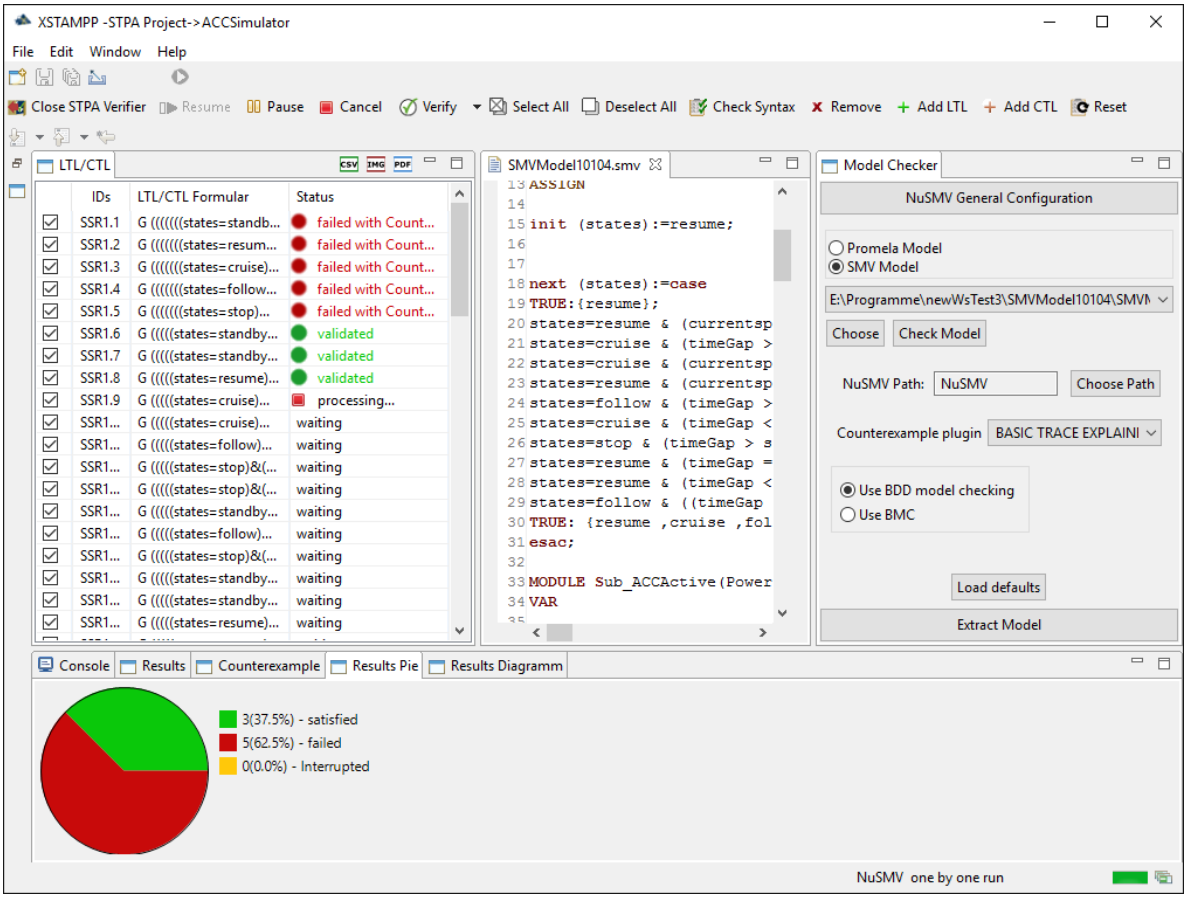


Abbildung 5.4: Die Oberfläche des STPA Verifiers im NuSMV Modus während einer Sicherheitsverifikation des ACCSimulators

Die Modelle wurden jeweils durch Ausführen einer automatisierten Verifikation mit dem STPA Verifier auf sämtliche Sicherheitsanforderungen überprüft wobei die Ergebnisse in Form einer Resultats-Tabelle wie sie in [AW15a] vorgeschlagen wurde und eines Kuchendiagramms sowie einer Darstellung aller ermittelter Werte in einem Prozent/Verifikations-Diagramm dokumentiert.

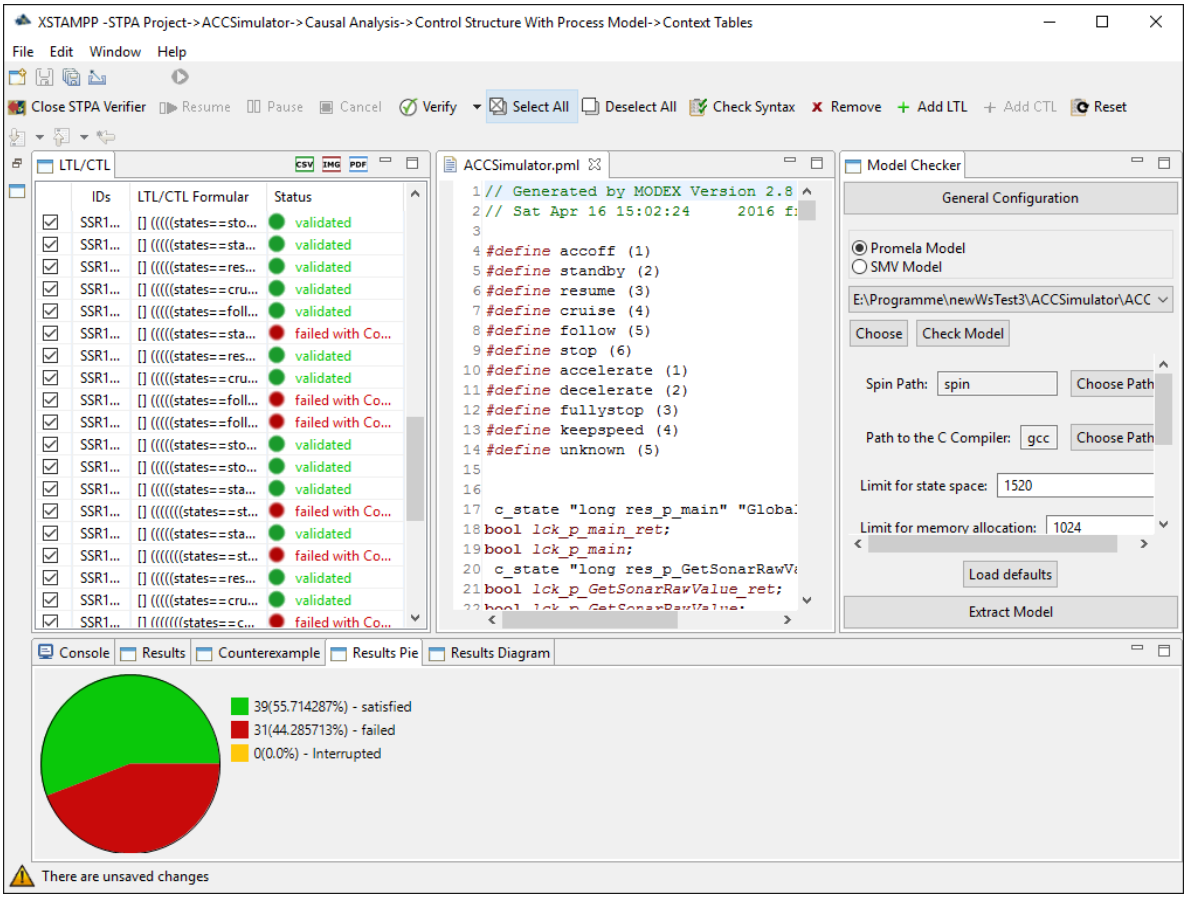


Abbildung 5.5: Beispielhafte Darstellung der Ergebnisse einer Sicherheitsverifikation des Promela Modells des ACCSimulator mit Spin 6.4.5 von 70 Anforderungen als Kuchendiagramm

Abbildung 5.5 zeigt die Ergebnisse der Verifikation aller in STPA erfassten Sicherheitsanforderungen. Durch die Darstellung als Kuchendiagramm lässt sich in Echtzeit der Trend der Verifikationen erkennen, der hier sehr schnell erkennen lässt das fast die Hälfte aller Verifikationen des Promela Modells ein Gegenbeispiel hervorgebracht haben.

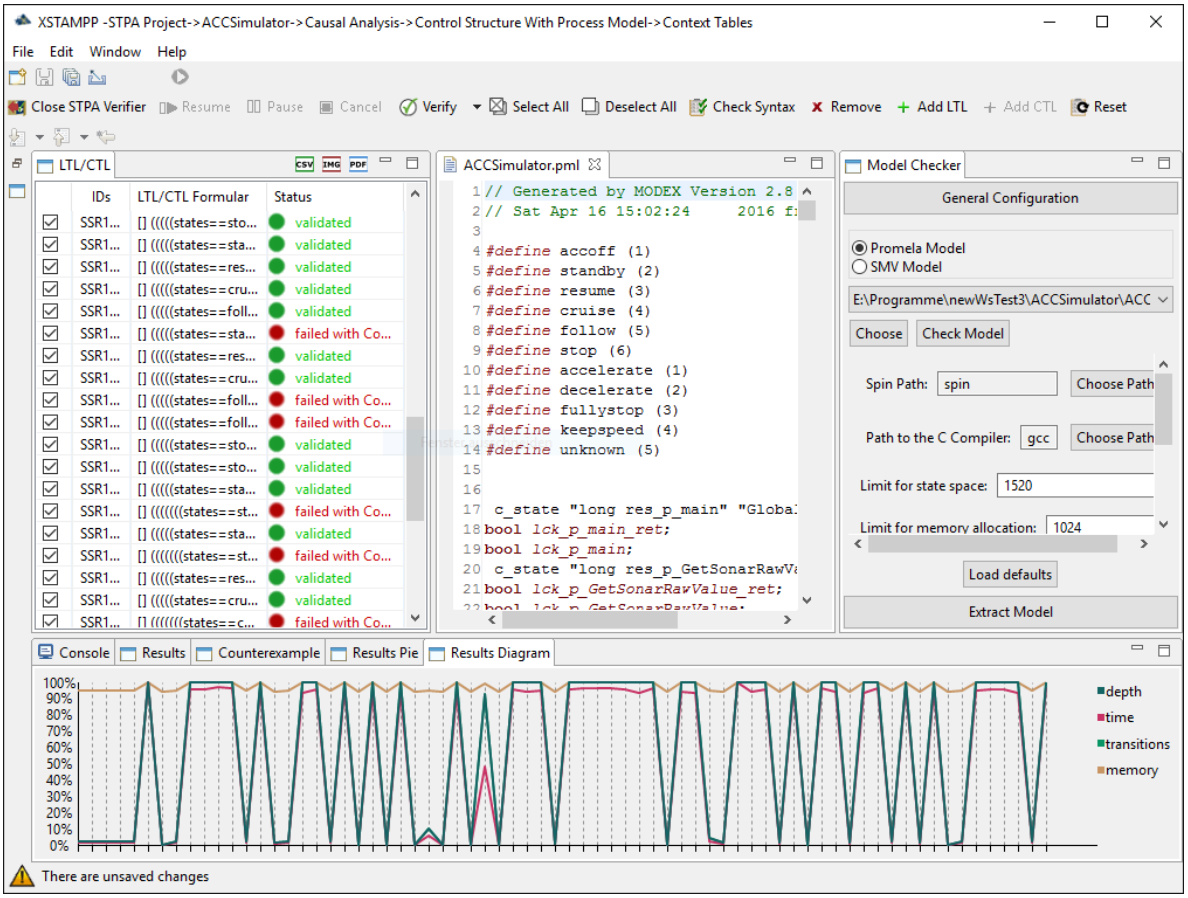


Abbildung 5.6: Darstellung der Ergebnisse einer Sicherheitsverifikation des Promela Modells des ACCSimulator mit Spin 6.4.5 von 70 Anforderungen mittels des Resultats Diagramms welches ein Prozent/Zeit Diagramm der Verifikations Werte darstellt

Results Table

SSR	#Depth	#Stored States	#Transitions	#Time	#Memory Usage	Result
SSR1.1	419430.0	9.95997E13	1.70086E8	0.02	1.389624E7	failed
SSR1.2	419430.0	9.95997E13	1.70086E8	0.01	1.389624E7	failed
SSR1.3	419430.0	9.95997E13	1.70086E8	0.02	1.389624E7	failed
SSR1.4	419430.0	9.95997E13	1.70086E8	0.02	1.389624E7	failed
SSR1.5	419430.0	9.95997E13	1.70086E8	0.02	1.389624E7	failed
SSR1.6	419430.0	9.95997E13	1.70086E8	0.01	1.3707184E7	satisfied
SSR1.7	419430.0	9.95997E13	1.70086E8	0.01	1.3707184E7	satisfied
SSR1.8	419430.0	9.95997E13	1.70086E8	0.01	1.3707184E7	satisfied
SSR1.9	419430.0	9.95997E13	1.70086E8	0.01	1.3707184E7	satisfied

Abbildung 5.7: Ausschnitt aus der Resultats-Tabelle einer Sicherheitsverifikation des Promela Modells des ACCSimulator mit Spin 6.4.5 von 70 Anforderungen

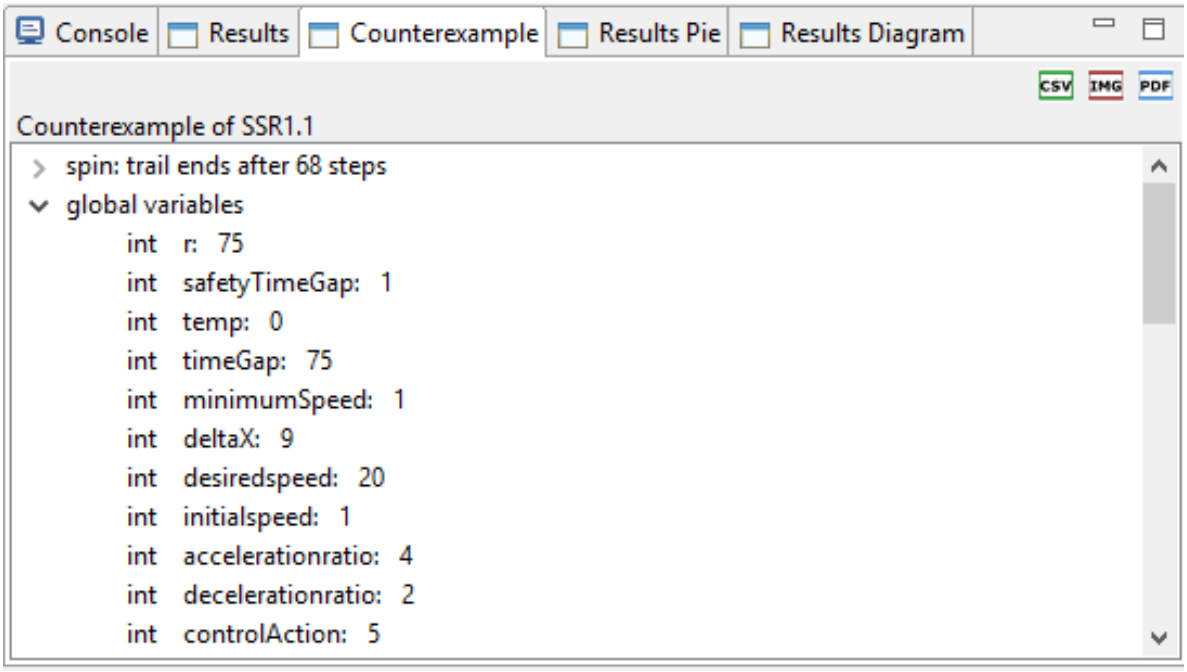


Abbildung 5.8: Beispielhafte Darstellung eines von Spin 6.4.5 berechneten Gegenbeispieles im STPA Verifier

6 Setup

In diesem Kapitel sollen die Systemvoraussetzungen und die Einrichtung des STPA Verifiers dargestellt werden.

6.1 Installation

Der STPA Verifier stellt folgende Anforderungen an das System:

- mindestens 1 GB RAM(empfohlen werden 2)
- 200 MB Festplattenspeicher (für XSTAMPP + STPA Verifier)
- empfohlen wird mindestens ein Zweikern-Prozessor (z.B. Intel Core i3)

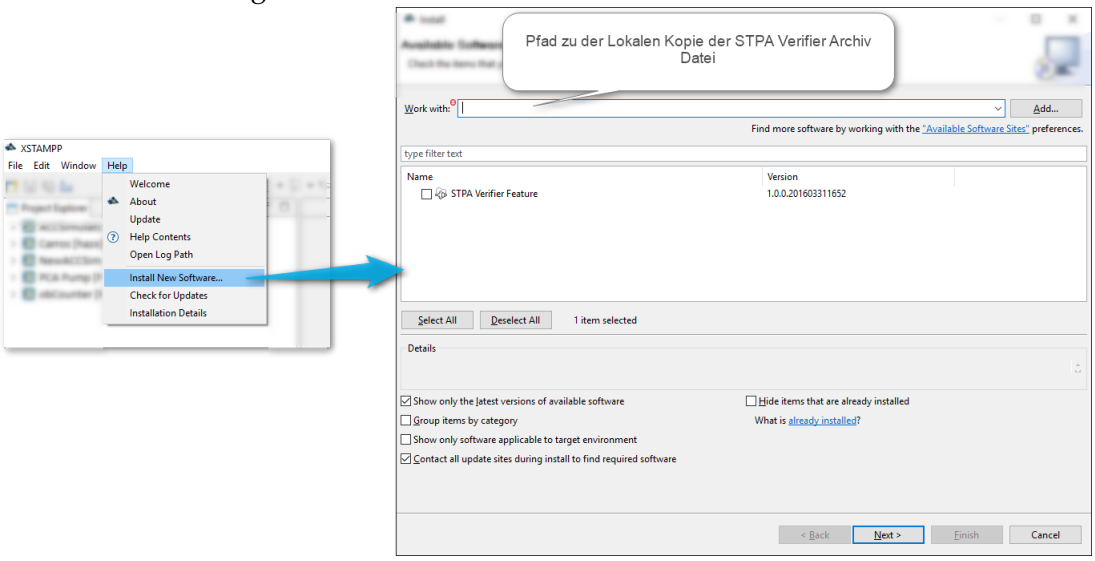
Des weiteren werden, um alle Funkionen des STPA Verifiers nutzen zu können, müssen die folgenden zusätzlichen Programme auf dem Rechner installiert sein:

- **Java 7 Runtime** [1]
- **C Compiler** (optional) (im Rahmen dieser Arbeit wurde der GCC[2] benutzt, es sind allerdings auch andere C Compiler möglich)
- **NuSMV** (optional) Es wird die zu diesem Zeitpunkt aktuelle Version 2.6.0 empfohlen, es wird allerdings mindestens NuSMV 2.0 benötigt da noch ältere Versionen kein BMC unterstützen.[3]
- **Spin** (optional) In dieser Arbeit wurde Version 6.4.5 des Model Checkers eingesetzt[4]
- **Modex** (optional) Um die Modex Einbindung in den STPA Verifier nutzen zu können muss Modex von[5] nach der auf der Seite vorhandenen Anleitung und den in Kapitel refchap:extraction gegebenen Anweisungen installiert werden.
- **XSTAMPP** Der STPA Verifier kommt als Plug-in für die XSTAMPP Platform Version 2.0.2 die von [6] bezogen werden.

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[2] https://gcc.gnu.org/
[3] Der NuSMV Model Checker kann unter http://nusmv.fbk.eu/NuSMV gedownloaded werden
[4] Der Spin Model Checker kann auf https://spinroot.org gedownloaded werden
[5] https://spinroot.org/extraction
[6] http://www.xstampp.de/Download.html

Das STPA Verifier Plug-in selber wird nach Beendigung dieser Arbeit mit auf der XSTAMPP Homepage unter dem Reiter *Tools* → *STPA Verifier* zum download bereitstehen. Zur Installation des STPA Verifiers müssen folgende Schritte durchgeführt werden:

1. **XSTAMPP Installieren** Hierfür muss die entsprechende Archiv Datei gedownloaded und dem gewünschten Installationsverzeichnis entpackt werden.
2. **Installation des STPA Verifiers**
   - 2.1. Download des STPA Verifier Update Archivs von[7]
   - 2.2. Installation des Plug-in's in XSTAMPP



[7]http://www.xstampp.de/Download.html

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Intention dieser Bachelorarbeit war es den Prozess der formalen Verifikation auf Softwareebene mit der Analyse von Sicherheitsanforderungen, durch STPA, auf Systemebene zu verbinden.
Zu diesem Zweck wurde im Fortgang dieser Arbeit der STPA Verifier implementiert und vorgestellt. Der STPA Verifier bietet eine grafische Oberfläche zur Konfiguration und Ausführung eines Model Checkers. Es wurde sowohl der Ansatz des symbolischen Model Checkings durch NuSMV als auch des expliziten Model Checkings durch Spin integriert, wodurch Anforderungen sowohl in LTL als auch CTL in einer geleiteten Verifikation geprüft werden können.

7.2 Ausblick

Dieser Abschnitt soll das Resümee dieser Arbeit aufzeigen und auf einige Eigenheiten und Limitierungen der einzelnen vorgestellten Methoden hinweisen und diese diskutieren. Außerdem soll ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben werden.

7.2.1 Limitierungen des *Modex* Werkzeugs

Wie bereits im Anwendungsbeispiel erwähnt traten einige Eigenheiten und Limitierungen von Modex bei der Anwendung auf den ACCSimulator auf. Die größten Schwierigkeiten mit dem durch Modex abgeleiteten Promela Modell war das Modex Promela fremde Variablen (Double,Float,usw.) nicht in regulär definierte Promela Variablen übersetzt. Stattdessen werden diese Variablen mit dem Codewort **c_ state** direkt in das Modell übernommen, wodurch sie zwar von dem durch Spin erzeugten Verifikationsprogramm aber nicht von Spin selber lesbar sind. Durch diese Einschränkung ist eine Verifikation dieser Variablen durch die Übergabe einer LTL Formel nicht direkt möglich.
Es werden zwei Lösungsansätze vorgeschlagen:

- Das in Kapitel 5 vorgestellte Beispiel wurde mittels einer modifizierten Version des ACCSimulator.c Programms durchgeführt, bei der für jede Promela fremde Variable ein Integer hinzugefügt wurde welches den Wert der Orginalvariable mit Integer-Genauigkeit mitschreibt. Ein solcher Ansatz ist in sofern praktikabel als das Integer problemlos durch Modex in Promela Integer übersetzt werden. Allerdings ist eine solche Manipulation des Quellcodes, gerade bei großen Systemen, extrem aufwändig und fehleranfällig.
- Abdulkhaleq und Wagner haben eine LTL Verifikation vorgestellt[1] die mittels logischer Aussagen, welche in Promela mit dem Codewort **c_ expr** definiert wurden, die **c_ state** Variablen auswertet:

  $$\#define\ pc\_expr\{Pp_controlSpeed->frontDistance <= now.safeDistance\}$$
  $$\#define\ qc\_expr\{now.accOperation == accelerate\}$$

  Diese Aussagen können dann in der LTL Formel $[](p \rightarrow q)$ verifiziert werden.

7.2.2 Future work

In dieser Arbeit wurde eine Oberfläche zur geführten Verifikation von Sicherheitsanforderungen vorgestellt welche die Model Checker Spin und NuSMV nutzt. Dabei wurde lediglich der für diese Arbeit definierte Leistungsumfang berücksichtigt. Beide Model Checker bieten allerdings noch zusätzliche Funktionen die in zukünftigen Arbeiten in den STPA Verifier integriert werden könnten. Vor allem Spin bietet mannigfaltige Möglichkeiten sowohl den Spin Prozessor als auch dem C Compiler oder dem ausführbaren Verifikationsprogramm sehr viele Spezialisierungen und Optimierungen beim Aufruf zu übergeben. Auch bieten beide beide Model Checker die Möglichkeit einer geführten Simulation des Zustandsraumes.

[1] https://github.com/asimabdulkhaleq/STPA-and-Software-Model-Checking

Literaturverzeichnis

[AW14a] A. Abdulkhaleq, S. Wagner. A-STPA: An Open Tool Support for System-Theoretic Process Analysis. *2014 STAMP Conference at Massachusetts Institute of Technology (MIT), 27 March 2014, Boston, USA.*, 2014. (Zitiert auf Seite 25)

[AW14b] A. Abdulkhaleq, S. Wagner. A Software Safety Verification Method Based on System-Theoretic Process Analysis. In *Computer Safety, Reliability, and Security*, S. 401–412. Springer, 2014.

[AW15a] A. Abdulkhaleq, S. Wagner. Integrated safety analysis using systems-theoretic process analysis and software model checking. In *Computer Safety, Reliability, and Security*, S. 121–134. Springer, 2015. (Zitiert auf den Seiten 8, 14, 21, 24, 26, 43 und 51)

[AW15b] A. Abdulkhaleq, S. Wagner. XSTAMPP: An eXtensible STAMP Platform As Tool Support for Safety Engineering. *2015 STAMP Conference at Massachusetts Institute of Technology (MIT), 26 March 2015, Boston, USA*, 2015. (Zitiert auf Seite 25)

[AW16] A. Abdulkhaleq, S. Wagner. XSTAMPP 2.0: New Improvements to XSTAMPP Including CAST Accident Analysis and an Extended Approach to STPA. *2016 STAMP Conference at Massachusetts Institute of Technology (MIT), 21 March 2016, Boston, USA*, 2016. (Zitiert auf Seite 26)

[AWL15] A. Abdulkhaleq, S. Wagner, N. Leveson. A Comprehensive Safety Engineering Approach for Software-Intensive Systems Based on STPA. *Procedia Engineering*, 128:2–11, 2015. (Zitiert auf den Seiten 2, 8, 23, 24 und 30)

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003. (Zitiert auf den Seiten 14 und 19)

[BK08] C. Baier, J. KATOEN. Principles of Model Checking (Representation and Mind Series).[Sl], 2008. (Zitiert auf Seite 13)

[CC10] P. Cousot, R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. *Logics and Languages for Reliability and Security*, 25:1–29, 2010.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CGP+02] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. Integrating BDD-based and SAT-based symbolic model checking. In *Frontiers of Combining Systems*, S. 49–56. Springer, 2002. (Zitiert auf Seite 18)

[Div16] N. C. S. Division. Automated Combinatorial Testing for Software (ACTS), 2016. URL http://csrc.nist.gov/groups/SNS/acts/index.html. (Zitiert auf Seite 26)

[EMC09] J. Edmund M. Clarke. Model Checking VI Linear-Time Temporal Logic, 2009. URL http://www.cs.cmu.edu/~emc/15817-f09/lecture6.pdf. (Zitiert auf Seite 18)

[Gum] H. P. Gumm. Model Checking. (Zitiert auf Seite 19)

[Gum07] H. P. Gumm. Lineare Temporale Logik, 2007. (Zitiert auf Seite 13)

[HHS01] G. J. Holzmann, M. H Smith. Software model checking: extracting verification models from source code†. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001. (Zitiert auf Seite 21)

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279, 1997. (Zitiert auf Seite 17)

[Hol04] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, Band 1003. Addison-Wesley Reading, 2004. (Zitiert auf Seite 17)

[Hol07] G. J. Holzmann. Design and Validation of Computer Protocols. *Computer Protocols*, 2007. (Zitiert auf Seite 8)

[Hol16] G. J. Holzmann. Spin - Formal Verification, 2016. URL spinroot.com. (Zitiert auf Seite 18)

[Jia14] Y. Jia. *Resilient and Efficient Delivery over Message Oriented Middleware.* Dissertation, Queen Mary University of London, 2014.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, A. Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, S. 97–109. Springer, 1993. (Zitiert auf Seite 14)

[Lev04] N. Leveson. A new accident model for engineering safer systems. *Safety science*, 42(4):237–270, 2004. (Zitiert auf Seite 10)

[Lev11] N. G. Leveson. *Engineering a Safer World - Systems Thinking Applied to Safety*. Massachusetts Institute of Technologie, 2011. (Zitiert auf den Seiten 8, 10, 24 und 25)

[McM93] K. L. McMillan. *Symbolic model checking*. Springer, 1993. (Zitiert auf den Seiten 8 und 18)

[Muk97] M. Mukund. Linear-time temporal logic and Büchi automata. *Tutorial talk, Winter School on Logic and Computer Science, Indian Statistical Institute, Calcutta*, 1997. (Zitiert auf Seite 8)

[Roz11] K. Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011. (Zitiert auf Seite 13)

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science, Volume B*, S. 133–191, 1990. (Zitiert auf Seite 17)

[Tho13] J. Thomas. *Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis*. Dissertation, Massachusetts Institute of Technology, 2013. (Zitiert auf den Seiten 8, 11, 12, 23, 26 und 48)

[VW86] M. Y. Vardi, P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, S. 322–331. IEEE Computer Society, 1986.

[VW94] M. Y. Vardi, P. Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994. (Zitiert auf Seite 14)

Alle URLs wurden zuletzt am 18. 04. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift