Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Bachelorarbeit

Abstraktion kontinuierlichen Verhaltens zur modellbasierten Sicherheitsanalyse

Autor:

Ben Rabeler

18. Mai 2015

Betreuer:

Prof. Dr. Frank Ortmeier

Institut für verteilte Systeme
Lehrstuhl für Softwaretechnik

M.Sc. Tim Gonschorek

Institut für verteilte Systeme
Lehrstuhl für Softwaretechnik

Rabeler, Ben:
Abstraktion kontinuierlichen Verhaltens zur modellbasierten Sicherheitsanalyse
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2015.

Inhaltsangabe

Die Sicherheit von technischen System ist in vielen Gebieten wie der Luftfahrt oder im Straßenverkehr von elementarer Bedeutung für die Hersteller solcher Systeme wie für die Anwender. Ein Ansatz um die Sicherheit von diskreten Systemen zu zeigen, besteht darin, diese in einem diskreten Modell zu abstrahieren und anschließend mit einem Model Checker auf gefährdende Situationen zu überprüfen. Viele Szenarien in den Anwendungsbereichen wie der Luftfahrt oder dem Straßenverkehr beruhen jedoch auf kontinuierlichem Systemverhalten.

In dieser Arbeit wird ein Verfahren vorgestellt, das ein kontinuierliches System in diskrete Automaten abstrahiert, sodass diese Automaten mit einem symbolischen Model Checkers auf eine Sicherheitsspezifikation überprüft werden können. Das Verfahren wurde in Umgebung, in der diskrete Modelle erstellt und überprüft werden können, als Prototyp umgesetzt. Dieser dient dazu anhand von Experimenten Eigenschaften des Verfahrens herauszufinden.

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xi**

Algorithmenverzeichnis **xiii**

Quelltextverzeichnis **xv**

1 Einführung **1**
1.1 Motivation 1
1.2 Hintergrund 2
1.3 Zielstellung der Arbeit 3
1.4 Aufbau der Arbeit 4

2 Grundlagen **5**
2.1 Differentialgleichungen 5
2.1.1 Definition von Differentialgleichungen 5
2.1.2 Anfangswertproblem (Cauchy-Problem) 5
2.2 Endliche diskrete Automaten 6
2.3 Hybride Automaten 7
2.4 Spezifikation von Modelleigenschaften 10
2.4.1 Kripke-Struktur 10
2.4.2 Linear Temporal Logic 10
2.4.3 CTL* 11
2.5 Model Checking 12
2.6 SAML 12
2.7 VECS 13
2.8 Intervallarithmetik 13
2.9 Algorithmus zur syntaktischen Anpassung von Übergängen 14

3 Transformation eines hybriden Modells in ein diskretes Modell **17**
3.1 Eine Darstellung eines hybriden Automaten in einem diskreten Automaten 17
3.1.1 Modellierung der Zeit 17
3.1.2 Modellierung homogener Differentialgleichungen 18
3.2 Transformationsprozesse 18
3.2.1 Lösen der Differentialgleichung 19
3.2.2 Auflösen der Vergleiche in Übergangsbedingungen 20

3.2.3 Auflösung der Überlappung des Fehlerintervalls bei Übergängen 21
3.2.4 Erkennen von sich überschneidenden Übergangsbedingungen . 21
3.2.5 Zusammenfassung der Transformationsprozesse 23
3.3 Behandlung eines parametrisierten Modells 23
3.3.1 Kennzeichnung von Parametern im Ausgangsmodell 23
3.3.2 Auflösung von Parametern in den Differentialgleichungen . . . 24
3.3.3 Auswertung der Ergebnisse des Model Checker 25
3.3.4 Behandlung von indeterministischen Verhalten 25

4 Implementierung des Transformators 27
4.1 Aufbau eines hybriden Automat in SAML 27
4.1.1 Bilden des Zeitsystems in SAML 27
4.1.2 Modellierung homogener Differentialgleichungen 28
4.2 Umsetzung der Transformationsprozesse 28
4.3 Schnittstelle zwischen ODE-Solver und Transformer 29
4.3.1 Scilab 29
4.3.2 Schnittstellen ODE Solver und ODE Solution 30
4.4 Kommunikation von Modelchecker und VECS 31

5 Beschreibung des Verfahrens anhand von zwei Fallstudien 35
5.1 Punktförmige Zugbeeinflussung (Indusi) 35
5.1.1 Beschreibung 35
5.1.2 Ablauf der Transformationen 36
5.2 Adaptive Cruise Control 39
5.2.1 Beschreibung 39
5.2.2 Veranschaulichung der Änderung von Parametern im Modell . 41

6 Feststellung der Eigenschaften des entwickelten Verfahrens anhand von Experimenten 45
6.1 Testumgebung 45
6.2 Punktförmige Zugbeeinflussung 46
6.2.1 Vergleich mit anderen Ansätzen 46
6.2.2 Experiment zur Performance der Transformationsprozesse . . 47
6.3 Adaptive Cruise Control 49
6.3.1 Verhalten der Adaptive Cruise Control bei wechselnden Parametern 49
6.3.2 Vergleich mit iSAT 50

7 Bewertung des Verfahrens 53
7.1 Beurteilung der Eigenschaften 53
7.2 Verbesserung der Implementierung 54
7.2.1 Behandlung von Indeterminismen 54
7.2.2 Erstellen von Vergleichen in VECS 55
7.2.3 Differentialgleichungen in SAML 55
7.3 Ausblick 55
7.3.1 Fehleranalyse beim numerischen Lösen der Differentialgleichung 55
7.3.2 Neuberechnung bei überlappendem Fehlerintervall 56
7.3.3 Bounded Model Checking 57

8 Zusammenfassung **59**

A Anhang **61**
A.1 SAML Modell der Punktförmigen Zugbeeinflussung 61
A.2 Tranformiertes Modell der Punktförmigen Zugbeeinflussung 61
A.3 iSAT Modell der Punktförmigen Zugbeeinflussung 63
A.4 SAML Modell der Adaptive Cruise Control 63
A.5 iSat Modell der Adaptive Cruise Control 64

Literaturverzeichnis **65**

Abbildungsverzeichnis

1.1 Beispielbild der Adaptive Cruise Control 2
1.2 Ein gelöstes Modell kommuniziert mit dem Model Checker 3
1.3 Ein Transformer als Kommunikator zwischen ODE-Solver und Model Checker 4

2.1 Automat einer 2-mal-12-Stunden-Zählung 7
2.2 Hybrider Automat eines Thermostats 10

3.1 Automat zur Speicherung von zeitlichen Zuständen 18
3.2 Flussdiagramm der Transformationsschritte 18
3.3 Interface eines Lösungsprogramms für Differentialgleichungen 19
3.4 Nichtdeterministisches Verhalten der Parameter im Verlauf des Model Checking 25
3.5 Wechsel eines Parameters bei nichtdeterministischen Verhalten im Verlauf des Model Checking 26
3.6 Backtracking zum Start des Nichtdeterminismus 26

4.1 Sequenzdiagramm der Transformationsschritte 29
4.2 Datenstrukturen zur Anwendung und Speicherung der Differentialgleichungen als Klassendiagramm 31
4.3 Sequenzdiagramm der Abläufe in der Kommunikation zwischen Prozessen des Model Checking und der Transformation 33

5.1 Schematische Darstellung des automatischen Bremsverhaltens bei einer punktförmigen Zugbeeinflussung 35
5.2 Darstellung des Beispiels der punktförmigen Zugbeeinflussung als hybrider Automat 36
5.3 Darstellung des Verlaufs von $vel(t)$ und $pos(t)$ für das Indusi-Modell . 38
5.4 Darstellung der ACC-Abläufe in der Fallstudie 39
5.5 Hybrider Automat der Adaptive Cruise Control 40
5.6 Berechnete Position der beiden Fahrzeuge für Ausgangsbedingungen . 41
5.7 Berechnete Position der beiden Fahrzeuge bei erstem Bremsen 42
5.8 Crashverhalten der beiden Fahrzeuge nach Bremsen des vorausfahrende Fahrzeugs 43
5.9 Gesamter Ablauf der ACC-Fallstudie 44

6.1 Entwicklung der Transformationszeit für verschiedene Diskretisierungszeiten Δt 48

Tabellenverzeichnis

6.1 Vergleich der Laufzeiten und Ergebnisse beim Prüfen der Fälle A und B mit $\Delta t = 0,1$ 46
6.2 Laufzeiten (in Sekunden) des Indusi-Modells für verschiedene Diskretisierungsstufen Δt 47
6.3 Laufzeiten (in Sekunden) des ACC-Modells für verschiedene Diskretisierungsstufen Δt und maximalen Zeitpunkten t_n 50
6.4 Laufzeiten (in Sekunden) des ACC-Modells für iSAT und SAML/-NuSMV für verschiedene Bremsstufen ($\Delta t = 0.1$) 51

Liste der Algorithmen

1 Ein Algorithmus zur syntaktischen Anpassung von Zustandsübergängen 14

2 Ersetzen von Vergleichen mit ODE-Variablen durch zeitliche Abhängigkeit in der Funktion *replaceComparisonsWithTimeSteps* 20

3 Ersetzen von Vergleichen mit ODE-Variablen durch zeitliche Abhängigkeit in der Funktion *getTimedCondition* als Pseudocode 20

4 Funktion *handleErrorInTransitionConditions* zur Transformation eines Modell mit aufgelösten Differentialgleichungen 22

5 Funktion *getCriticalTransitions* zum Erkennen von sich überschneidenden Übergangsbedingungen 22

6 Funktion *isCritical* zum Testen ob alle Variablen innerhalb des Fehlerintervalls liegen 23

7 Funktion *evaluateComparisons* zur Auswertung von Vergleichen . . . 23

8 Pseudocode der Transformation in ein Modell mit aufgelösten Differentialgleichungen 24

9 Backtracking Verfahren 26

10 Funktion *handleErrorInTransitionConditions* zur Transformation eines Modell mit aufgelösten Differentialgleichungen 56

Quelltextverzeichnis

2.1 Beispiel der 2-mal-12-Stunden-Zählung als SAMl-Quelltext 13
2.2 Beispiel von drei beispielhaften Zustandsübergängen 14
2.3 Angepasste Zustandsübergänge in SAML-Code 15
3.1 Beispiel einer Zeitkomponente 18
4.1 Modellierung der globalen Zeit 27
4.2 Modellierung von Differentialgleichungen in SAML 28
4.3 Modellierung und Lösung eines Systems von Differentialgleichungen in Scilab 30
4.4 Kennzeichnung von Parametern in einem SAML-Modell 32
5.1 Transformierte Zeitkomponente 37
7.1 Verbesserte Modellierung von Differentialgleichungen in SAML 55

1. Einführung

In diesem Kapitel wird zuerst die Motivation hinter der Arbeit anhand einer Abstandskontrolle für Fahrzeuge erläutert. Darauf aufbauend wird erläutert, unter welchem Hintergrund die Arbeit entstanden ist. Im Anschluss werden die Ziele eingeführt und wie die Ziele erreicht werden sollen. Abschließend wird in diesem Kapitel ein Überblick über die Arbeit gegeben.

1.1 Motivation

Viele ingenieurwissenschaftliche Systeme werden mittlerweile durch Computer gesteuert. Diese Systeme haben sich aus der Sicht der Softwareentwicklung über die Jahre zu immer komplexeren Softwaresystemen entwickelt. Ein Bericht der NASA hat ergeben, dass für das Kampfflugzeug F-35, das 2006 seinen Erstflug hatte, 5,7 Millionen Zeilen Code geschrieben wurden, während bei der F-4, die 1960 eingeführt wurde, nur 1000 Zeilen Code verwendet wurden. In dem Bericht wird geschätzt, dass der Anteil der von Software bereitgestellten Funktionalität im Zeitraum von 1960-2000 von 8% auf 80% gestiegen ist [D+09, Seite: 1].

Im Design solcher Systems ergeben sich für die Softwareentwicklung verschiedene Herausforderungen. Eine Herausforderung besteht z.B. in der Sicherstellung, dass das System fehlerfrei funktioniert.

Um solche Eigenschaften eines derartig komplexen Systems herauszufinden oder zu überprüfen, werden im Entwicklungsprozess verschiedene Modelle von verschiedenen Sichtweisen auf das System erstellt.

Ein Beispielsystem für den möglichen Einsatz eines Modells ist die Adaptive Cruise Control [1]. Eine Skizze dieses Systems ist in Abbildung 1.1 dargestellt.

In dem Beispiel befinden sich zwei Fahrzeuge auf einer idealisierten Fahrbahn. Das bedeutet, dass die Fahrbahn eben ist und keine Reibungskräfte auf die Fahrzeuge wirken. Das Ziel der Abstandskontrolle besteht darin, dass das hinterherfahrende Fahrzeug seine Geschwindigkeit v_2 an die des vorfahrenden Fahrzeugs v_1 anpasst, sodass es in erster Linie zu keinem Zusammenstoß kommt [LCJ95].

Dafür misst es durch Sensoren den Abstand $p_2 - p_1$ und die Geschwindigkeit des vorherfahrenDasden Fahrzeugs. Anhand dieser Werte und Daten aus dem Auto, in

[1]Deutsch: Abstandsregelung, im Folgenden durch ACC abgekürzt

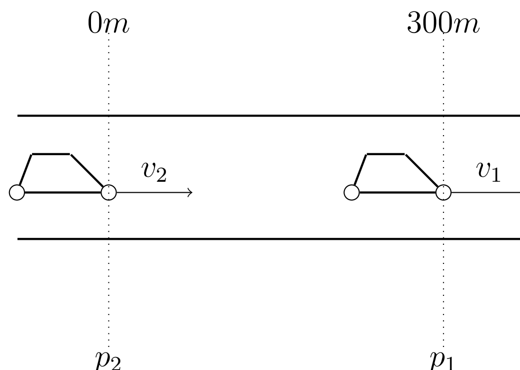

Abbildung 1.1: Beispielbild der Adaptive Cruise Control (angelehnt an [GdSMH01])

dem eine ACC eingesetzt wird, kann nun die Geschwindigkeit des hinterherfahrenden Fahrzeugs so angepasst werden, dass es dem vorfahrenden Fahrzeug folgt.

Diese Darstellung entspricht bereits einem einfachen Modell. Dieses könnte nun durch Werteparameter für Beschleunigung, Geschwindigkeit und Position der zwei (oder mehr) Fahrzeuge erweitert werden. Diese veränderlichen Werteparameter eines Modells werden in Natur- und Ingenieurwissenschaften mit Differentialgleichungen beschrieben [CC97].

Neben der Modellierung der Parameter als ein System von kontinuierlichen Differentialgleichungen bietet sich aber auch die Möglichkeit einen Parameter wie die Beschleunigung durch ein Zustandsdiagramm zu modellieren. Die Beschleunigung kann dabei mehrere Zustände (hohe Beschleunigung, niedrige Beschleunigung etc...) verfügen. Hierbei handelt es sich um ein zeitdiskretes Modell (also ein Modell, dass über die kontinuierliche Zeit diskrete Werte annimmt).

Für dieses Beispiel ist es sinnvoll, diese beiden Formen der Modellierung zu kombinieren. Einen solchen Zweck erfüllt ein hybrider Zustandsautomat (siehe Abschnitt 2.3). Dieser kombiniert das dynamische Verhalten von Parametern mit denen von diskreten Zuständen. Eine ACC könnte somit durch mehrere Zustände modelliert werden, die die Beschleunigung beschreiben. Innerhalb dieser Zustände werden Differentialgleichungen verwendet, die abhängig von der jeweiligen Beschleunigung die Position und Geschwindigkeit beschreiben.

Für ein solches Modell werden Sicherheitseigenschaften spezifiziert. Eine typische Spezifikation an die modellierte ACC wäre, ob für ein beliebiges Szenario das hinterherfahrende Fahrzeug seine Geschwindigkeit reduziert oder es sogar zu einem Zusammenstoß kommt.

Ob diese Spezifikation für ein Modell erfüllt ist, kann in der Entwicklung eines derartigen Systems sehr wichtig sein, weil das Bauen eines Prototyps (bei der ACC ein Fahrzeug) kostenintensiv ist.

Ebenso ist der Einsatz formaler Methoden für die Prüfung von Spezifikationen überlegenswert. Durch vereinzelte Tests für gewisse Aspekte des Systems können zwar Aussagen für diese Testfälle getroffen werden. Diese decken aber nicht alle möglichen Abläufe des Modells ab.

1.2 Hintergrund

Am Lehrstuhl für Softwaretechnik der Universität Magdeburg wurde die Modellierungssprache SAML (siehe Abschnitt 2.6) entwickelt. Ausgehend von dieser Sprache

wurde eine IDE (siehe Abschnitt 2.7) entwickelt, die in der Lage ist, SAML durch eine Übersetzung der Sprache in andere Sprachen an verschiedene Programme (Model Checker) anzubinden, die in der Lage sind eine gegebene Spezifikation an das Model zu überprüfen. Ein Problem, dass sich im Kontext der Modellierung darstellt ist, dass SAML nur diskrete und endliche Zustandsautomaten modellieren kann, wodurch ein dynamisches System wie das der ACC nur über Umwege modelliert werden kann.

In der Vergangenheit wurden bereits Verfahren entwickelt, um Differentialgleichungen mit der Modellierung von diskreten Zeitschritten in SAML anzubinden. Im Modell wurde dafür ein numerisches Lösungsverfahren eingesetzt, das eine numerische Lösung der Differentialgleichung beschreibt. Dieses Vorgehen ist in Abbildung 1.2 dargestellt.

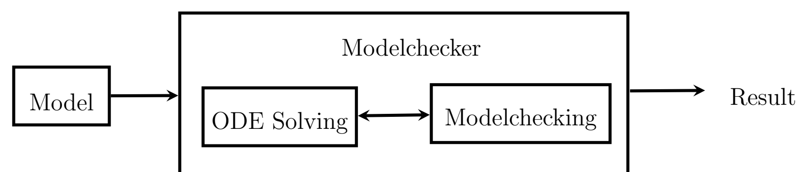

Abbildung 1.2: Ein gelöstes Modell kommuniziert mit dem Model Checker

In [Nie13] wurde ein Verfahren vorgestellt, das innerhalb eines beliebigen symbolischen Model Checker die Differentialgleichung mithilfe eines numerischen Algorithmus gelöst hat. Das Verfahren hat sich für sehr Modelle mit wenigen Diskretisierungsstufen als anwendbar herausgestellt, insofern dass es für eine Differentialgleichung für manche Fälle sicher feststellen kann, dass das Modell unsicher ist. Bei größeren Modellen hat sich jedoch herausgestellt, dass abhängig von den gewählten Zeitschritten der entstandene Fehler zu groß wurde (Zeitschritte zu groß gewählt) oder der Zustandsraum zu groß wurde (Zeitschritte zu klein gewählt).

Unabhängig der Einbettung in SAML gibt es bereits Lösungsmöglichkeiten, um ein System wie die ACC modellieren zu können. Hierfür wurden hybride Automaten entwickelt und Algorithmen um solche Automaten auf Spezifikationen zu überprüfen. Diese Algorithmen lösen Differentialgleichung für ein Fehlerintervall. Sofern dieses Intervall überschritten ist, beginnen diese eine neue Berechnung des Modells. Ein Problem dieser Technik ist für große Modelle, dass das Modell sehr häufig neu berechnet wird [HHWT97].

1.3 Zielstellung der Arbeit

Das Ziel der Arbeit ist ein Verfahren zu entwickeln, das Differentialgleichungen aus einem hybriden Modell mithilfe eines numerischen Lösungsprogramm für Differentialgleichungen löst, anschließend die Lösung der Differentialgleichung in das ursprüngliche Modell einfügt und das Modell auf eine gegebene Spezifikation mit einem symbolischen Model Checker überprüft. Der Ansatz ist in Abbildung 1.3 dargestellt.

Dieser Ansatz soll hinsichtlich seiner Eigenschaften und der Anwendbarkeit untersucht werden.

Dabei werden bzgl. Modell und Differentialgleichungen folgende Einschränkungen getroffen:

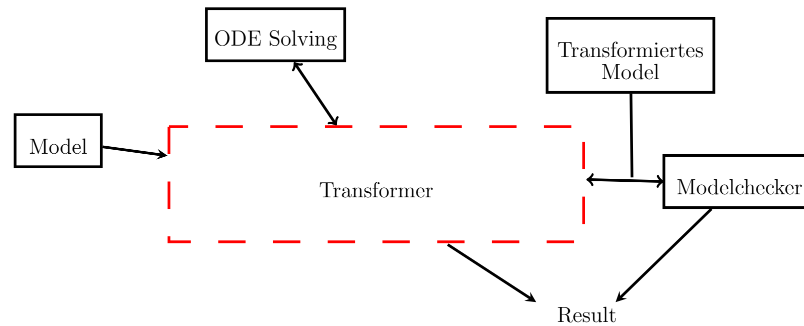

Abbildung 1.3: Ein Transformer als Kommunikator zwischen ODE-Solver und Model Checker

1. Es werden nur gewöhnliche Differentialgleichungen 1.Ordnung der Form $\dot{x} = f(x, t)$ betrachtet. Die Differentialgleichungen dürfen sich während der Ausführung des Modells nicht ändern.
2. Die Modelle sind zeitlich beschränkt.
3. Es wird angenommen, dass die Sicherheitseigenschaft $|\boldsymbol{AG}\phi \Leftrightarrow \neg\boldsymbol{EF}\neg\phi|$ (siehe Abschnitt 2.4) gilt, also nur Erreichbarkeitsprobleme betrachtet werden.

Das vorgestellte Verfahren soll folgende Zwischenschritte erfüllen:

1. Numerisches Lösen der Differentialgleichung mithilfe eines externen numerischen Programms.
2. Transformation eines hybriden Modells (siehe Abschnitt 2.3) in ein SAML-Modell unter Beirücksichtung der gelösten Differentialgleichung.
3. Behandlung des entstehenden Fehlers beim Lösen der Differentialgleichung.

Wenn diese Umsetzung erfolgt ist, soll eine Bewertung der Qualität des Verfahrens stattfinden. Dieses soll anhand der Berechnungsdauer und Belegung von Ressourcen wie Arbeitsspeicher für verschieden komplexe Modelle festgemacht werden. Außerdem soll der Ansatz mit einem weiteren hybriden Model Checker verglichen werden.

1.4 Aufbau der Arbeit

Die Arbeit ist in sechs Teile gegliedert. Im ersten Teil werden Algorithmen, Begriffe, Notationen eingeführt, die im späteren Verlauf der Arbeit verwendet werden. Der zweite Teil stellt eine Möglichkeit vor, mit der hybride Automaten in einer Sprache wie SAML modelliert werden können. Anschließend wird in diesem Teil ein Verfahren vorgestellt, dessen Aufgabe es ist, ein System von Differentialgleichung in einem zeitlich begrenzten Intervall zu lösen und die gefundene Lösung in das gegebene Modell einzufügen.

Im dritten Teil wird eine Umsetzung des vorher vorgestellten Verfahrens eingeführt. Der vierte Teil erläutert die Abläufe des Verfahrens anhand von zwei Fallstudien, sodass anhand dieser Fallstudien im fünften Teil Experimente zur Feststellung der Eigenschaften der umgesetzten Implementierung festgemacht werden können. Abschließend wird im letzten Teil das Verfahren bezüglich Stärken und Schwächen analysiert.

2. Grundlagen

In diesem Kapitel werden verschiedene Konzepte und Notationen, die im Rahmen dieser Arbeit benutzt werden, eingeführt. Die ersten Abschnitte lassen sich unabhängig von vorherigen Abschnitten lesen. Die letzten Abschnitte, in dem die Sprache SAML und anschließend ein Algorithmus, der SAML verwendet, eingeführt werden, greifen auf die eingeführten Notationen oder Beispiele vorheriger Abschnitte zurück.

2.1 Differentialgleichungen

In diesem Abschnitt soll zuerst eine Definition eines Differentialgleichungssystems eingeführt werden. Anschließend wird das Anfangsproblem eingeführt und unter welchen Bedingungen dieses lösbar ist.

2.1.1 Definition von Differentialgleichungen

Der Definition von [Aul97] folgend ist mit einer Funktion $F : D \to \mathbb{R}$ eine (N-dimensionale gewöhnliche) Differentialgleichung (n-ter Ordnung) definiert durch

$$F(t, x, \dot{x}, \ldots, x^{n-1}, x^n = 0 \tag{2.1}$$

mit den Variablen $t \in \mathbb{R}$ und $x, \dot{x}, \ldots, x^{n-1}, x^n \in \mathbb{R}^N$. Es wurde in [Aul97] gezeigt, dass sich ein N-dimensionales System n-ter Ordnung auf ein System 1.Ordnung reduzieren lässt. Im Folgenden werden deshalb nur noch (N-dimensionale) Systeme 1.Ordnung betrachtet, die folgende Form besitzen:

$$\dot{\mathbf{x}}(t) = \begin{pmatrix} f_1(t, \mathbf{x}) \\ f_2(t, \mathbf{x}) \\ \vdots \\ f_N(t, \mathbf{x}) \end{pmatrix} \tag{2.2}$$

2.1.2 Anfangswertproblem (Cauchy-Problem)

Bei dem Anfangswertproblem ist ein die Lösung zu einem bestimmten Zeitpunkt t_0 bekannt [Aul97]. Das Gleichungssystem

$$\dot{x} = f(t, x) \quad , \quad x(t_0) = x_0 \tag{2.3}$$

wird als Anfangswertproblem bezeichnet [BHWM85]. x_0 ist dabei der Anfangsvektor und besitzt folgende Form:

$$x_0 = \begin{pmatrix} x_1(t_0) \\ x_2(t_0) \\ \dots \\ x_i(t_0) \\ \vdots \\ x_n(t_0) \end{pmatrix} \tag{2.4}$$

Zur numerischen Lösung von Anfangsprobleme können verschiedene numerische Verfahren verwendet werden. Eine Übersicht und Einführung in diverse Verfahren bietet [SWP12].

Eine Voraussetzung für Lösbarkeit ist, dass die Funktion f auf dem Streifen

$$S := \{(t, x) : t_0 \leq t \leq t_e, x \in \mathbb{R}^n\}$$

auf dem Intervall Lipschitz-stetig ist. In diesem Fall ist die Differentialgleichung nach dem Satz von Picard-Lindelöf auf dem Intervall $[t_0, t_{end}]$ eindeutig lösbar [SWP12].

Für die Lipschitz-Stetigkeit von f ist hinreichend, dass f in einer offenen Menge $D \supset S$ stetig differenzierbar bezüglich x ist [SWP12].

2.2 Endliche diskrete Automaten

Ein (nichtdeterministischer) diskreter Automat ist nach [Sch01] spezifiziert durch das 5-Tupel

$$(Z, \Sigma, \delta, S, E)$$

Hierbei sind:

- Z eine endliche Menge von Literalen (Zustände)
- Σ eine endliche Menge (Eingabealphabet), es gilt $Z \cap \Sigma = \emptyset$
- δ eine Funktion $Z \times \Sigma$ nach $P(Z)$ (Überführungsfunktion)
- $s_0 \in Z$ der Startzustand
- $E \subseteq Z$ die Menge der Endzustände (optional)

Wenn E definiert ist, spricht man von einem endlichen Automaten. Sofern δ eine totale Funktion ist und es keine Überführung in eine Menge von Zuständen, sondern in einzelne Zustände existiert, spricht man von einem deterministischen (endlichen) Automaten [Sch01].

In Abbildung 2.1 ist ein Automat als Graph dargestellt, der eine Uhr modelliert, die nach 2-mal-12-Stunden-Zählung 12 Zeitzustände (für die Stunden) anzeigen kann sowie ob es Vormittag (a.m.) oder Nachmittag (p.m.) ist. Zustände werden hier durch Kreise beschrieben. Die Zustandsmenge Z ergibt sich dementsprechend als $Z = \{(c = 0, am), (c = 0, pm), \dots, (c = 11, am), (c = 11, pm)\}$. Eine Eingabe wird hier als Label auf einer Kante dargestellt. Das Eingabealphabet ist für dieses Beispiel gegeben durch $\Sigma = \{am, pm, !am, !pm\}$. Zu beachten ist, dass der Zustand

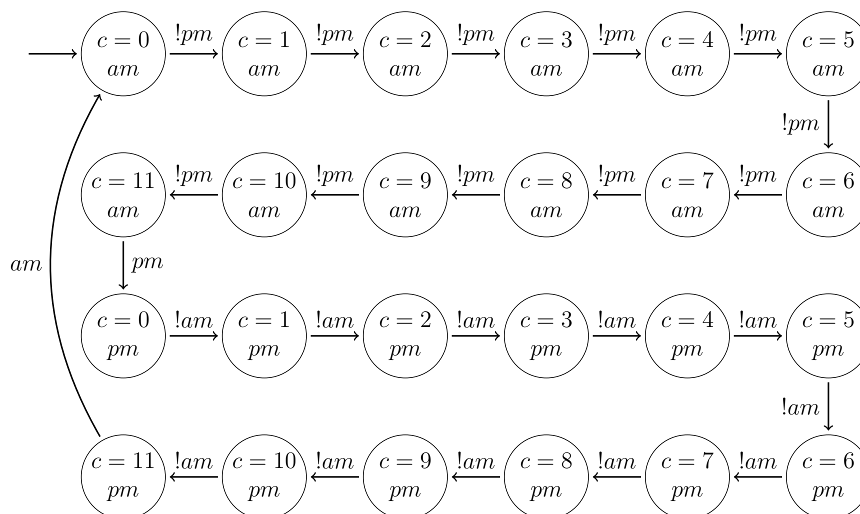

Abbildung 2.1: Automat einer 2-mal-12-Stunden-Zählung

am unterschiedlich zum Übergang ist. Der Automat besitzt in diesem Beispiel ein wilkürlich gewählten Startzustand mit $s_0 = (c = 0, am)$. In diesem Beispiel ist kein Endzustand definiert.

Die Zustandsübergangsfunktion ist definiert durch

$$\begin{aligned}
\delta((c = 0, am), !pm) &= \{(c = 1, am)\} \\
&\vdots \\
\delta((c = 11, am), pm) &= \{(c = 0, pm)\} \\
\delta((c = 0, pm), !am) &= \{(c = 1, pm)\} \\
&\vdots \\
\delta((c = 11, pm), am) &= \{(c = 0, am)\}
\end{aligned}$$

Dies bedeutet, dass der Folgezustand immer der nächste zeitliche Schritt ist. Sofern ein Überlauf (11 zu 12) stattfindet, wird der Zeiger für Uhrzeit auf 0 zurückgesetzt und das Signal (am/pm) wird getauscht.

2.3 Hybride Automaten

Ein hybrider Automat ist eine veränderter endlicher Automat, mit dem dynamisches Verhalten innerhalb eines Zustands genauer modelliert werden kann. Mit den hybriden Automaten wird das diskrete Verhalten aus indeterministischen diskreten Automaten mit dem dynamischen Verhalten von naturwissenschaftlichen System kombiniert [MS00].

Wie bereits in dem vorherigen Beispiel der 2-mal-12-Stunden-Zählung ist es für ausgewählte Probleme sinnvoll, sich innerhalb eines Zustands bestimmte Parameter abzuspeichern und bei verändertem Parametern einen Übergang des Zustands in einen anderen Zustand durchzuführen. Dafür ist es notwending das Verhalten des Automat innerhalb eines Zustands näher zu definieren wie auch das Zustandsübergangsverhalten auf diese Änderungen anzupassen.

Im folgenden ist ein hybrider Automat spezifiziert, Diese Spezifikation ist angelehnt an [MS00] und [Att01]. Ein hybrider Automat ist ein 8-Tupel

$$(X, V, inv, init, flow, E, upd, jump)$$

mit:

- X ist eine endliche Menge $\{x_1, x_2, \ldots, x_n\}$ von Variablen, die über $\mathbb{R}$ definiert sind. Eine Belegung s ist ein Punkt aus $\mathbb{R}^n$. Der Wert einer Variable x_i ist die i-te Komponente einer Belegung $s \in s_i$. Für ein Prädikat Ψ über die Variablen aus X schreiben wir $[[\Psi]]$, um die Menge aller Belegungen zu kennzeichnen, für die $\Psi[X := s] = true$ gilt.

- V ist eine endliche Menge von Stellen. Ein Zustand eines hybriden Automat ist gekennzeichnet durch ein Tupel (v, s) mit $v \in V$ und einer Belegung s.

- inv legt Invarianten fest, die für eine Stelle gelten müssen. Dies ist eine Abbildung $V \rightarrow \Psi$, d.h. jeder Stelle wird ein Prädikat über Variablen aus X zugeordnet.

- $init$ ist eine Zuordnung der Menge aller Stellen auf Prädikate von Variablen aus X. $init(v)$ wird als Startbedingung einer Stelle bezeichnet. Der Zustand (v, s) ist genau dann ein initialer Zustand des Automat, wen $init(v)$ und $inv(v)$ für s wahr sind.

- $flow$ (Flow-Invariante) ist eine Zuordnung von einer Stelle aus V zu Prädikaten $X \cup \dot{X}$ mit $\dot{X} = \{\dot{x_1}, \ldots, \dot{x_n}\}$. $\dot{x_i}$ bezeichnet dabei gewöhnliche Differentialgleichungen 1.Ordnung.

  Mit dieser Abbildung können nun Zustandsübergänge beschrieben werden, für die die aktuelle Stelle gleich bleibt, die internen Variablen des Automat sich jedoch ändern. Für eine Zeitschritt-Relation $\stackrel{\delta}{\rightarrow}$ existiert der Übergang $(v, s) \stackrel{\delta}{\rightarrow} (v, s')$ genau dann, wenn es eine differenzierbare Funktion $f : [0, \delta] \rightarrow R$ gibt und folgende Bedingungen erfüllt sind:

  - $f(0) = s$ und $f(\delta) = s'$
  - die Invariante von v ist erfüllt $f(t) \in [[inv(v)]]$ für $t \in [0, \delta]$
  - das Prädikat $flow$ ist erfüllt: $flow(v)[X, \dot{X} := f(t), \dot{f}(t)]$ ist wahr für $t \in [0, \delta]$, mit $\dot{f} = (\dot{f_1}(t), \ldots, \dot{f_n}(t)$

- $E \subseteq V \times V$ ist eine Multimenge von Kanten (Übergängen). Ein Übergang $(v, v') \in E$ hat die Startstelle v und die Zielstelle v'.

- upd legt für jeden Übergang eine „Update-Menge“ fest. Dies bedeutet, dass diejenigen Variablen $v \in X$ sich verändern können, wenn der Übergang benutzt wird. Formal handelt es sch um eine Abbildung von E auf eine Menge M für die gilt: $M \subseteq X$

- $jump$ ist eine Abbildung von E auf Sprungbedingungen. Die Abbildung beschreibt, welcher Übergang stattfinden kann und welche Variablen neue Werte

zugewiesen werden. Die Sprungbedingung $jump(e)$ ist ein Prädikat über die Variablen aus X. Bei einem Zustandsübergang vergeht keine Zeit.

Die Übergangsrelation $\rightarrow$ ist definiert als $(v, s) \xrightarrow{\delta} (v', s')$ genau dann, wenn:

- $e = (v, v')$
- die Invarianten $s \in [[inv(v)]]$ und $s \in [[inv(v')]]$ erfüllt sind
- die Variablen in $upd(e)$ ändern sich wie in der Updatebedingung definiert: $jump(e)[X, upd'(e) = s, s'[upd(e)]]$ ist wahr und $s'[upd(e)]$ bezeichnet die Variablen aus s', die in $upd(e)$ enthalten sind.
- die Variablen, die nicht in $upd(e)$ enthalten sind, bleiben unverändert: $s_i = s'_i$ für alle $x_i \in X \setminus upd(e)$

In Abbildung 2.2 wird ein einfaches Thermostat als hybrider Automat modelliert. Das Beispiel ist [Hen00] entnommen.

Für das Beispiel lassen sich folgende graphischen Elemente aus der Abbildung lassen sich der formalen Definition zuordnen:

- Kreise stellen Stellen dar. Der Name einer Stelle wird in den Kreis geschrieben, hier: $V = \{on, off\}$
- Eine Invariante $inv(v)$ einer Stelle v wird durch ein Prädikat (z.B. $inv(off) = x < 19$) innerhalb einer Stelle gekennzeichnet.
- Die Flow-Invariante einer Stelle ist gekennzeichnet durch ein Prädikat, das eine Differentialgleichung beschreibt (z.B. $flow(off) = (\dot{x} = -0.1x)$.
- Die Variablenmenge $X = \{x\}$ lässt sich aus allen Invarianten ablesen.
- Die Sprungbedingungen sind durch Zustandsübergangspfeile mit der jeweiligen Updatebedingung gekennzeichnet. In diesem Beispiel ist der Fall nicht modelliert, dass sich eine Variable mit einem Update ändert. Dieser Fall wird wie eine Updatebedingung über den Pfeil notiert (z.B. mit $x := 0$ an einem Pfeil).
- Die Startbedingung ist wie bei einem endlichen Automaten durch einen Pfeil auf die zugehörige Stelle gekennzeichnet.
- Die Zustände (v, s) des hybriden Automat sind somit die Kreise gekennzeichnet, in denen die Stellen modelliert sind sowie durch die möglichen Variablenbelegungen innerhalb der Knoten.

Im Folgenden wird mit dem Begriff Parameter eine Variable bezeichnet, die in keiner Flow-Invariante vorkommt. Ein hybrider Automat, für den alle Variablen Parameter sind, wird als parametrisierter Automat bezeichnet. Der in Abbildung 2.1 dargestellte Automat der 2-mal-12-Stunden-Zählung könnte z.B. als parametrisierter Automat interpretiert werden. Die Uhrzeit sowie die Belegung (am/pm) sind hier die Parameter, die sich von Zustand zu Zustand über Updateregeln ändern.

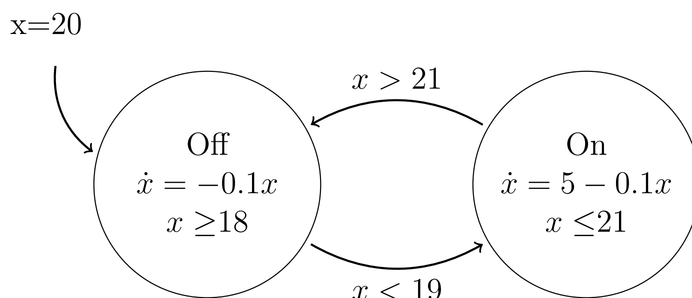

Abbildung 2.2: Hybrider Automat eines Thermostats

2.4 Spezifikation von Modelleigenschaften

Zur Betrachtung des Verhaltens von Zustandssystem werden die Pfade der Zustandsübergänge betrachtet. Um diese spezifizieren zu können, ist ein formales Hilfsmittel nötig. In dieser Arbeit wird Computation Tree Logic (CTL) als Logik verwendet.

Dazu wird zuerst die Kripke-Struktur eingeführt anhand der CTL* eingeführt wird. Aus dieser lassen sich Logiken LTL und CTL ableiten.

2.4.1 Kripke-Struktur

Eine Kripke-Struktur dient zur Beschreibung eines Zustandssystems.

$$\tau_{kripke} = (S, S_0, T, L, AP)$$

Mit:

- S als endliche, nicht leere Menge von Zuständen
- $S \subseteq S$ als Menge der initialen Zustände
- $T \subseteq S \times S$ als totale Übergangsrelation. Das bedeutet, dass für jeden Zustand $s \in S$ ein Zustand $s' \in S$ existiert, sodass $(s, s') \in T$
- $L : S \to 2^{AP}$ ist eine Abbildung, die jedem Zustand $s \in S$ eine Teilmenge der atomaren Ausdrücke

Der Pfad π einer Kripke-Struktur ist definiert durch eine Sequenz von Zuständen $s_i \in S$

$$\pi = s_0, s_1, s_2, \dots$$

2.4.2 Linear Temporal Logic

Die Linear Time Logic (LTL) wurde in [Pnu77] 1977 eingeführt, um Aussagen über das Verhalten auf Pfaden treffen zu können [Sch02] [BA12].

LTL hat als Ziel temporale Ereignisse in eine Logik zu fassen. Betrachtet wird in dieser Logik ein Pfad von Zuständen $\psi = s_1, s_2, \dots$. Für jeden Zustand s_i kann eine Eigenschaft $phi \in \{wahr, falsch\}$ gelten.

Die Logik führt im Folgenden mehrere Operatoren ein, die zu wahr oder falsch ausgewertet werden und somit in die Syntax der Aussagenlogik integriert werden können:

- **G** ϕ (global), für alle Zustände auf dem betrachteten Pfad gilt ϕ
- **F** ϕ (finaly), irgendwann auf dem betrachteten Pfad gilt ϕ

Es gibt mit **U** (until) und **X** (next) zwei weitere Operatoren, die LTL ebenso verwendet. Da diese Arbeit keinen dieser Operatoren wird auf eine genauere Einführung verzichtet.

2.4.3 CTL*

CTL* ist eine Erweiterung von LTL [Sch02, Seite 9]. Das Ziel besteht darin, nicht ausschließlich das Verhalten von einem Pfad zu betrachten, sondern auch eine Verzweigung (Branch) betrachten zu können. Dafür führt die Logik die Modaloperatorén **A** (für alle Pfade) und **E** (für einen Pfad) ein.

Die folgende Syntaxdefintion von CTL* ist [Gü11] entnommen und entspricht der ursprünglich in [EH86] eingeführten Definition:

- Wenn $p \in AP$, dann ist p eine Zustandsformel
- Wenn ϕ_i und ϕ_j Zustandsformeln sind, dann sind auch $\neg\phi_i$, $\phi_i \wedge \phi_j$ und $\phi_i \vee \phi_j$ Zustandsformeln
- Wenn ϕ eine Pfadformel ist, dann sind **E** ϕ und **A** ϕ Zustandsformeln
- Wenn ϕ eine Zustandsformel ist, dann ist ϕ auch eine Pfadformel
- Wenn ϕ_i und ϕ_j Pfadformeln sind, dann sind auch $\neg\phi_i$, $\phi_i \wedge \phi_j$, $\phi_i \vee \phi_j$, $\mathbf{X}\ \phi_i$, $\mathbf{G}\ \phi_i$ und $\phi_i\ \mathbf{U}\ \phi_j$ Pfadformeln

Nach [EH86] ist die Semantik von CTL* durch folgende induktive Definition gegeben. Dabei ist τ_{Kripke} eine Kripke-Struktur, ϕ bezeichnet eine Zustandsformel und ψ eine Pfadformel. π^i entspricht dem Suffix eines Pfads π, der im Zustand s_i beginnt.

$$
\begin{aligned}
\tau_{Kripke}, s \models p &\Leftrightarrow p \in L(s) \\
\tau_{Kripke}, s \models \neg\phi &\Leftrightarrow \tau_{Kripke}, s \not\models \phi \\
\tau_{Kripke}, s \models \phi_1 \wedge \phi_2 &\Leftrightarrow \tau_{Kripke}, s \models \phi_1 \textit{ und } \tau_{Kripke}, s \models \phi_2 \\
\tau_{Kripke}, s \models \phi_1 \vee \phi_2 &\Leftrightarrow \tau_{Kripke}, s \models \phi_1 \textit{ oder } \tau_{Kripke}, s \models \phi_2 \\
\tau_{Kripke}, s \models \mathbf{E}\psi &\Leftrightarrow \text{es existiert ein Pfad } \pi \text{ von } s \text{, sodass } \tau_{Kripke}, \pi \models \psi \\
\tau_{Kripke}, s \models \mathbf{A}\psi &\Leftrightarrow \text{für jeden Pfad } \pi \text{ von } s \text{ gilt } \tau_{Kripke}, \pi \models \psi \\
\tau_{Kripke}, \pi \models \phi &\Leftrightarrow s \text{ ist der erste Zustand von } \pi \text{ und } \tau_{Kripke}, s\pi \models \phi \\
\tau_{Kripke}, \pi \models \neg\psi &\Leftrightarrow \tau_{Kripke}, \pi\neg \models \psi \\
\tau_{Kripke}, \pi \models \psi_1 \wedge \psi_2 &\Leftrightarrow \tau_{Kripke}, \pi \models \psi_1 \textit{ und } \tau_{Kripke}, \pi \models \psi_2 \\
\tau_{Kripke}, \pi \models \psi_1 \vee \psi_2 &\Leftrightarrow \tau_{Kripke}, \pi \models \psi_1 \textit{ oder } \tau_{Kripke}, \pi \models \psi_2 \\
\tau_{Kripke}, \pi \models \mathbf{F}\psi &\Leftrightarrow \exists i \geq 0 : \tau_{Kripke}, \pi^i \models \psi \\
\tau_{Kripke}, \pi \models \mathbf{G}\psi &\Leftrightarrow \forall i \geq 0 : \tau_{Kripke}, \pi^i \models \psi
\end{aligned}
$$

Da CTL* in Pfadformeln und Zustandsformeln unterscheidet, lassen sich mit CTL (Computation Tree Logic) und LTL zwei weiterverbreitete Logiken als Untermenge von CTL* definieren. In CTL muss jedem Modaloperator einer der bereits vorgestellten Temporaloperatoren folgen. In LTL darf kein Modaloperator benutzt werden [Gü11].

2.5 Model Checking

Model Checking hat zum Ziel, zu prüfen, ob eine gegebene Spezifikation für ein Model zutrifft. Model Checking wurde in [CE82] und [QS82] in 1982 eingeführt, wofür die Autoren 2007 den Turing-Preis erhielten. Seitdem haben sich verschiedene Techniken zum Überprüfen etabliert.

Beim symbolischen Modelchecking handelt es sich um eine Technik, die alle Eigenschaften des Modells binär codiert und dann die gegebene Spezifikation über ein Binary Decision Diagram (BDD) auflöst [BCM+90].

Eine Alternative, mit der ein Pfad mit LTL-Formeln überprüft werden kann, bietet Bounded Model Checking (BMC). Ein Bounded Model Checker erhält als Eingabe eine Schrittweite k. Anhand dieser kann das Model Checking auf ein Erreichbarkeitsproblem (SAT) reduziert werden. Somit können bekannte SAT-Solver eingesetzt werden [Bie09] [BCC+03].

Ein Beispiel für einen Model Checker ist NuSMV. NuSMV kann sowohl symbolisches Modelchecking als auch Bounded Model Checking anwenden. Das Ergebnis ist sofern ein fehlerhaftes Ergebnis für eine Spezifikation gefunden wurde, den Pfad, auf dem der Fehler gefunden wurde [CCG+02].

Für hybride Modelle existieren ebenso Model Checker, die das das Ziel haben ein Erreichbarkeitsproblem zu lösen. Einer dieser Model Checker ist iSat [1]. iSat verfügt über einen Bounded Model Checking Ansatz, indem solange eine Grenze k gesucht wird, bis das Erreichbarkeitsproblem gelöst ist. Für jeden Berechnungsdurchgang k wird dabei das System neu berechnet [FHT+07].

Eine Übersicht über die den Aufbau eines iSAT-Programms bietet [iDT10].

2.6 SAML

SAML (System Analysis And Modeling Language) ist eine Modellierungssprache, die in [Gü11] eingeführt wurde. Sie dient dazu sicherheitskritische Modelle modellieren zu können. Das Ziel von SAML ist eine Abstraktionsebene zwischen weitverbreiteten Model Checkern und Modellierungssprachen wie UML oder Modelica herstellen, sodass diese Modelle mithilfe der Model Checker auf einfache Weise auf Sicherheitseigenschaften überprüft werden können.

Im Wesentlichen ist ein SAML-Modell ein diskreter, nichtdeterministischer, endlicher Zustandsautomat. SAML erlaubt probalistische Betrachtungen, in dem für jeden Zustandsübergang Wahrscheinlichkeiten angegeben werden.

In Quelltext 2.1 wird der in vorgestellte Automat als SAML-Modell modelliert.

Der Automat wird durch eine Komponente CLOCK umfasst. Die Zustände des Automaten werden durch ganzzahle Bereiche definiert und müssen jeweils eine Startbelegung besitzen. Für das Beispiel wird die Angabe am/pm durch den Zustand t modelliert, der die Werte 0 oder 1 annehmen kann und die Startbelegung 0 für am besitzt.

Zustandsübergänge sind durch Zustandsübergangsbedingungen (Conditions) und dem im nächsten Schritt erreichten Zielzustand (Assignment) gekennzeichnet. Eine wichtige Eigenschaft in SAML zur Anbindung an einen symbolischen Model Checker wie NUSMV ist, dass die Übergangsbedingungen für alle möglichen Kombinationen an möglichen Zuständen einen Zustandsübergang bereitstellen müssen.

[1] Weitere Informationen auf: https://projects.informatik.uni-freiburg.de/projects/isat/

Quelltext 2.1: Beispiel der 2-mal-12-Stunden-Zählung als SAMl-Quelltext

```
component CLOCK
  c : [0..11] init 0;
  t : [0..1] init 0;

  c < 11 & t = 0 -> c' = c + 1 & t' = 0;
  c = 11 & t = 0 -> c' = 0 & t' = 1;
  c < 11 & t = 1 -> c' = c + 1 & t' = 1;
  c >= 11 & t = 1 -> c' = 0 & t' = 0;
endcomponent
```

2.7 VECS

VECS[2] - Verification Environment for Critical Systems - ist eine Integrierte Entwicklungsumgebung zum Erstellen und Bearbeiten von SAML-Modellen. VECS bietet die Möglichkeit SAML-Modelle in gängige Sprachen von Model Checkern wie PRISM oder NuSMV zu übersetzen und anschließend auf eine Spezifikation zu überprüfen [LSO12a] [LSO12b]. Außerdem sind erste Interfaces zu Modellierungssprachen wie UML oder Modelica entstanden [Gon13].

2.8 Intervallarithmetik

Die Lösung einer Differentialgleichung für einen bestimmten Zeitpunkt wird in dieser Arbeit als ein Intervall betrachtet. Sollte eine Bedingung Berechnungen von mehreren Werten beinhalten, ist es nötig, eine Berechnungsvorschrift für Intervall einzuführen. In dieser Arbeit wird in diesem Fall die Intervallarethmetik verwendet wie sie u.a. in [Kea96] spezifiziert wurde.

Seien mit $\mathbf{x} = [x_i, x_s]$ und $\mathbf{y} = [y_i, y_s]$ zwei abgeschlossene Intervalle gegeben, dann gilt für eine Operation:

$$\mathbf{x} \circ \mathbf{y} = \{x \circ y | x \in \mathbf{x} \wedge y \in \mathbf{y}\} \tag{2.5}$$

Dies bedeutet, dass alle Fälle einer Operation abgedeckt werden können und mit der Berechnung von Minimum und Maximum einer Operation das nachfolgende Intervall bestimmt werden kann.

Für die Standardoperationen $+, -, \cdot$, ergeben sich die folgenden Rechenvorschriften:

$$\mathbf{x} + \mathbf{y} = [x_i + y_i, x_s + y_s] \tag{2.6}$$

$$\mathbf{x} + \mathbf{y} = [x_i - y_i, x_s - y_s] \tag{2.7}$$

$$\mathbf{x} \cdot \mathbf{y} = [\min x_i y_i, x_i y_s, x_s y_i, x_s y_s, \max x_i y_i, x_i y_s, x_s y_i, x_s y_s] \tag{2.8}$$

$$\frac{1}{\mathbf{y}} = [\frac{1}{y_s}, \frac{1}{\mathbf{y_i}}] \quad y_s \neq 0 \wedge y_i \neq 0 \tag{2.9}$$

$$\mathbf{x} \div \mathbf{y} = \mathbf{x} \cdot \frac{1}{\mathbf{y}} \tag{2.10}$$

[2]Weitere Informationen auf: https://cse.cs.ovgu.de/vecs/

2.9 Algorithmus zur syntaktischen Anpassung von Übergängen

Sofern ein Variable x aus einer Zustandsübergangsbedingung ϕ durch ein Intervall beschrieben wird, kann es zu dem Fall kommen, dass $x \geq \pi$ und $x < \pi$ gilt.

Sollte diese Variable innerhalb von mehreren Zustandsübergängen auftreten, muss ein Weg gefunden werden, aus dieser Menge von nicht-disjunkten Zustandsübergängen, eine Menge an disjunkten Zustandsübergängen zu finden.

Dafür wurde in [Nie13] ein Algorithmus eingeführt, der nicht-disjunkte Zustandsübergänge anpasst. Der Algorithmus ist hier in Algorithmus 1 dargestellt.

Algorithm 1 Ein Algorithmus zur syntaktischen Anpassung von Zustandsübergängen

input : Zustandsübergangsbedingungen $R_1, R_2, \ldots, R_n$, die eine oder mehrere Variablen x eines Differentialgleichungssystems verwenden
input : Ziele der Zustandsübergangsbedingungen $A_1, A_2, \ldots, A_n$
output : Eine Menge U an Zustandsübergängen

1 $M^{n\times n^2} \leftarrow \begin{pmatrix} 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & \ldots & 0 & 1 \\ 0 & 0 & \ldots & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & 1 & \ldots & 0 & 1 \\ 1 & 1 & \ldots & 1 & 1 \end{pmatrix}$

```
2  for i ← 0 to n do
3      Condition ← ∅
4      Assignment ← ∅
5      for j ← 0 to n do
6          if M_ij = 0 then
7              Condition ← Condition ∧¬R_i
8          else
9              Condition ← Condition ∧R_i
10             Assignment ← addNotDeterminismTOAssign(Assignment, A_i)
11     if Assignment ≠ ∅ then
12         U ← U∪ addNotDeterminismTOAssign(Condition, Assignment)
13 return U
```

Der Algorithmus soll im Folgenden anhand eines Beispiels demonstriert werden. Dafür seien in Quelltext 2.2 $n = 3$ Zustandsübergänge in SAML-Code gegeben:

Quelltext 2.2: Beispiel von drei beispielhaften Zustandsübergängen

```
x > a -> i' = 1;
x < a -> i' = 2;
x = a -> i' = 3;
```

In diesem Beispiel könnte es dazu kommen, dass für ein Intervall a mehrere der Zustandsübergänge wahr werden. Das Ziel des Algorithmus ist es, das Modell so anzupassen, dass alle Übergangsmöglichkeiten betrachtet werden. Wenn z.B. $x \geq a$ gilt, dann soll das Modell sowohl den Fall betrachten, dass i im nächsten Schritt 1 oder 3 ist.

Im ersten Schritt wird die Matrix M konstruiert:

$$M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Die Spalten der Matrix entsprechen den Zustandsübergängen. Diese werden in diesem Beispiel mit u_1, u_2, u_3 bezeichnet. Die Zeilen entsprechen den zu konstruierenden angepassten Zustandsübergängen.

Zur Konstruktion dieser Übergänge wird über jede Zeile i der Matrix iteriert. Sofern $M_{ij} = 0$, wird zur bisher für die Zeile erstellten Zustandsbedingung die negierte Bedingung u_i mit einem logischen Und angefügt. Analog wird für $M_{ij} = 1$ die Bedingung ohne Negation angefügt, jedoch ein Indeterminismus für das Übergangsziel des Übergangs u_i eingefügt.

Ausgenommen der ersten Zeile, in der für alle $M_{ij} = 0$, werden die neu konstruierten Zustandsübergänge zurückgegeben. Für das Beispiel der drei Zustandsübergänge ergibt sich nun folgendes angepasstes Modell:

Quelltext 2.3: Angepasste Zustandsübergänge in SAML-Code

```
!x > a & !x < a & x = a -> i' = 3;
!x > a & x < a & !x = a -> i' = 2;
!x > a & x < a & x = a -> choice: i' = 2 + choice: i' = 3;
x > a & !x < a & !x = a -> i' = 1;
x > a & !x < a & x = a -> choice: i' = 1 + choice: i' = 2;
x > a & x < a & !x = a -> choice: i' = 1 + choice: i' = 3;
x > a & x < a & x = a ->
   choice: i' = 1 + choice: i' = 2 + choice: i' = 3;;
```

3. Transformation eines hybriden Modells in ein diskretes Modell

Dieses Kapitel führt einen Transformationsprozess ein, der ein hybrides System mithilfe eines externen Lösungsprogramms für Differentialgleichungen für ein endliches Zeitintervall auflöst und alle Übergänge, die von Werten der Differentialgleichungen unter Berücksichtigung des entstandenen Fehlers anpasst.

Abschließend wird betrachtet, wie Differentialgleichungen zu betrachten sind, die von einem oder mehreren Parametern abhängig sind.

3.1 Eine Darstellung eines hybriden Automaten in einem diskreten Automaten

Diser Abschnitt beschreibt, welche Möglichkeiten es gibt, um einen hybriden Automaten (wie in Abschnitt 2.3 eingeführt) in SAML zu definieren. Dazu muss zunächst die Zeit näher definiert werden.

3.1.1 Modellierung der Zeit

Momentan besitzt SAML kein eigenständiges Zeitsystem, weshalb eine Zeit separat modelliert werden muss. Eine Möglichkeit dazu ist die Einführung einer Zeitkomponente. Der Automat einer 2-mal-12-Stunden-Zählung aus Abbildung 2.1 hat ein ähnliches Verhalten, da der Automat für jeden Berechnungsschritt immer eine Stunde hinzufügt. Das gewünschte Ziel ist jedoch ein Automat, der seine Zeit solange erhöht bis er eine vorher definierte Grenze erreicht hat.

In Abbildung 3.1 ist ein derartiger Automat dargestellt. Die Zeit wird hier beginnend mit einer Startzeit t_0 (START) als endliche Menge von Zuständen aufgefasst. Über jeden Schritt, den der Automat ausführt, wird die Zeit um einen festen Wert Δt erhöht bis letztlich die Grenze der Zeit t_{end} erreicht wird. In diesem Automaten ist $timeover := (TIME + \Delta t >= t_{end})$.

Dieser Automat lässt sich wie bereits vorgestellt in ein SAML-Modell umformen. Eine mögliche Umsetzung ist in Quelltext 5.1 dargestellt.

Mit einer solchen Zeitkomponente können unter der Voraussetzung, dass zwischen jedem Zyklus im Modell immer dieselbe Zeit vergeht, zeitlich beschränkte Modelle mit einer festen Zeitdifferenz Δt erstellt werden. Zu beachten ist, dass wenn in

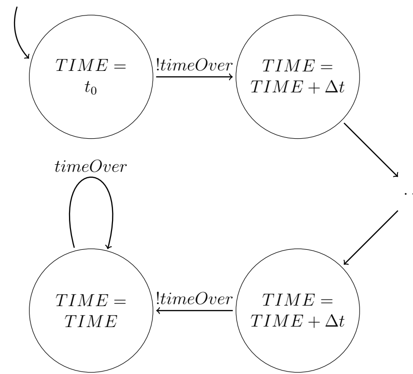

Abbildung 3.1: Automat zur Speicherung von zeitlichen Zuständen

Quelltext 3.1: Beispiel einer Zeitkomponente

```
component CLOCK
  TIME : [START..END] init START;
  TIME + 1 < END -> TIME' = TIME + deltaTime;
  TIME + 1 >= END -> TIME' = TIME;
endcomponent
```

SAML die Zeit als ein derartiger Automat modelliert wird, alle Übergänge im Modell angepasst werden, sodass sie nur gültig sind, solange der letzte Zeitpunkt erreicht wurde.

3.1.2 Modellierung homogener Differentialgleichungen

Im zugrundeliegenden Modell M muss es möglich sein, ein n-stelliges System von Differentialgleichungen, wie es in Abschnitt 2.1.1 definiert wurde, zu deklarieren. Für den folgenden Abschnitt wird angenommen, dass dies Differentialgleichungen der Form $x = f(x,t)$ sind, diese also von Zeit abhängig sein können, aber von keinen weiteren Parametern.

3.2 Transformationsprozesse

Im Folgenden wird ein Algorithmus zur Transformation eines in SAML definierten Modells mit gewöhnlichen Differentialgleichungen vorgestellt. Der Algorithmus ist als Flussdiagramm in Abbildung 3.2 dargestellt.

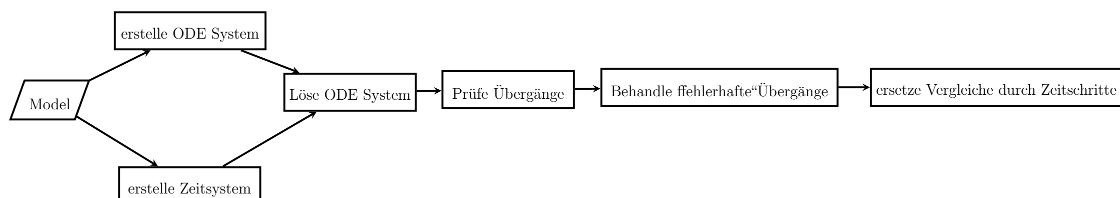

Abbildung 3.2: Flussdiagramm der Transformationsschritte

Als Eingabe wird ein Modell vorausgesetzt, das einen, wie im vorherigen Abschnitt beschriebenen, hybriden Automaten enthält.

Der Algorithmus ist in fünf Teile gegliedert. Der erste Teil behandelt die initiale Behandlung des Eingangsmodells. In dieser Phase wird, wie im vorherigen Abschnitt

beschrieben wurde, aus dem Eingangsmodell ein Zeitsystem und ein System von Differentialgleichungen extrahiert. Parallel dazu werden die Definitionen aller Differentialgleichungen gelöscht und eine Transformation des Zeitsystems durchgeführt.

Im zweiten Teil wird das System gelöst. Die nähere Ausführung der Prozesse werden in Abschnitt 3.2.1 beschrieben. Der dritte und vierte eil behandelt den Umgang mit dem bei der Lösung der Differentialgleichungen entstehenden Fehler und dessen Auswirkung auf die Übergangstransitionen im Automaten. Im fünften Teil wird die Lösung der Differentialgleichung in das Modell durch Ersetzen der Vergleiche in die Übergangsbedingungen gebracht. Da Teil drei zur Behandlung der fehlerhaften Transitionen auf dem Auflösen der Vergleiche in den Übergangsrelationen aufbaut, werden zuerst Teil drei in Abschnitt 3.2.2 behandelt, während abschließend in Abschnitt 3.2.3 die Behandlung der fehlerhaften Übergangstransitionen erklärt wird.

3.2.1 Lösen der Differentialgleichung

In dieser Arbeit wird angenommen, dass ein Interface zur numerischen Lösung eines Systems von gewöhnlichen Differentialgleichungen existiert.

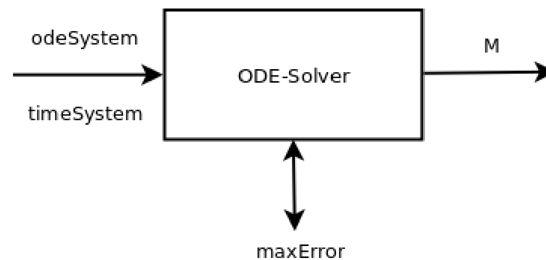

Abbildung 3.3: Interface eines Lösungsprogramms für Differentialgleichungen

In Abbildung 3.3 ist dargestellt, welche Eingaben und welche Ausgaben ein solches Interface bieten muss. Die Eingaben sind ein System aus Differentialgleichungen, das als Vektor $\mathbf{v}$ der Länge m und einem Zeitsystem (wie in Abschnitt 3.1 eingeführt).

Die Ausgabe ist eine Matrix $M^{n \times m}$. Dabei ist m die Anzahl der vorkommenden Zeitschritte, die sich mit t_0 als Beginn t_{end} als definiertem Ende und Δt als Intervalllänge des Zeitsystems aus der folgenden Formel ergeben:

$$n = \frac{t_{end} - t_0}{\Delta t} \tag{3.1}$$

Im Folgenden wird mit t_n der letzte Zeitschritt bezeichnet.

Die Lösungsmatrix M enthält nun die Lösungswerte für die einzelnen betrachteten Zeitschritte (Spalten) und die betrachteten Differentialgleichungen (Zeilen):

$$M_{ij} = \begin{pmatrix} x_1(t_0) & x_1(t_1) & \dots & x_1(t_i) & \dots & x_1(t_n) \\ x_2(t_0) & x_2(t_1) & \dots & x_2(t_i) & \dots & x_2(t_n) \\ \vdots & \vdots & & \vdots & & \vdots \\ x_j(t_0) & x_j(t_1) & \dots & x_j(t_i) & \dots & x_j(t_n) \\ \vdots & \vdots & & \vdots & & \vdots \\ x_n(t_0) & x_n(t_1) & \dots & x_n(t_i) & \dots & x_n(t_n) \end{pmatrix} \tag{3.2}$$

Da beim numerischen Lösen eines Differentialgleichungssystems pro Zeitschritt ein Fehler entsteht, wird angenommen, dass das Interface in der Lage ist, den Fehler für alle Zeitschritte garantiert unter einem maximalen Fehlerschwellwert ϵ zu halten.

3.2.2 Auflösen der Vergleiche in Übergangsbedingungen

Nachdem eine Lösung des Differentialgleichungssystems gefunden wurde, ist es das Ziel, die gefundenen Lösungswerte in das Modell einzufügen. Dafür müssen im Modell alle Übergangsbedingungen betrachtet werden, die von der Differentialgleichung abhängen. Das Finden der zu betrachtenden Übergangsbedingungen ist in folgendem Algorithmus 2 dargestellt:

Algorithm 2 Ersetzen von Vergleichen mit ODE-Variablen durch zeitliche Abhängigkeit in der Funktion *replaceComparisonsWithTimeSteps*

```
input  : The Model H
input  : The Solved System M
input  : A Set of all Transitions TRANS which have to be checked
14 for ∀tr_i, tr_i ∈ TRANS do
15     COMP = getComparisons(transition_i)
16     for ∀c_i, c_i ∈ COMP do
17         if hasODEIdentifier(c_i) then
18             condition ← getTimedCondition(comparison_I, M)
19             replaceComparisonWithCondition (H, c_i, condition
```

Es wird über alle Übergänge und dann alle alle Vergleiche iteriert. Anschließend werden die Vergleiche mit x und y als Terme der Form $x \simeq y$ mit $\simeq \in \{<, >, \leq, \geq, =\}$ betrachtet, in denen gilt, dass in x oder y eine Variable enthalten ist, die im Differentialgleichungssystem verwendet wird. Wenn ein solcher Vergleich gefunden wird, wird die Funktion *getTimedCondition* anwendet, die im folgenden Algorithmus 3 dargestellt ist:

Algorithm 3 Ersetzen von Vergleichen mit ODE-Variablen durch zeitliche Abhängigkeit in der Funktion *getTimedCondition* als Pseudocode

```
input  : A comparison c, The solved System M
output : A Condition Expression with subtituated ODE Identifiers
20 condition ← new
21 for ∀t_i do
22     c' ← copy(c)
23     for x_j ∈ getODEIdentifier(c') do
24         replaceIdentifierWithSolution(c', x_j, M_{t_i,x_j}, t_i)
25     evaluateTempComparison(c')
26     appendOr(condition)
27     condition ← condition+c'
28     appendAnd(condition)
29     condition ← condition + getTimeReference(t_i)
30 return condition
```

In dem Pseudocode wird von t_0 bis t_n iteriert. Für jeden Zeitschritt wird neben der eingefügten Lösung für den Zeitschritt eine Zeitreferenz eingefügt, sodass für einen Vergleich $x_i \simeq a$ mit $a \in \mathbb{R}$ und x_i als Variable, die in einem System von Differentialgleichungen benutzt wird, in einem Zeitbereich von t_0 bis t_n folgende Bedingung entsteht:

$$t_0 \wedge (M_{x_i,0} \simeq a) \vee t_1 \wedge (M_{x_i,1} \simeq a) \vee \cdots \vee t_n \wedge (M_{x_i,n} \simeq a) \tag{3.3}$$

Für diese Arbeit wird angenommen, dass ein Vergleichsterm, der eine ODE-Variable enthält, keine Referenz auf eine weitere Variable enthalten darf. Unter dieser Annahme bestehen die Vergleichsterme nach der Anwendung des Ersetzungsalgorithmus ausschließlich als Zahlenwerten und können ausgewertet werden (Anwendung der Funktion $evaluateTempComparison$ im Pseudocode), sodass nach dieser Auswertung ein Term entsteht, der, wenn der Vergleich $x_i \simeq a$ für t_i, t_j und t_k wahr ist, folgende Form besitzt:

$$t_i \vee t_j \vee t_k \tag{3.4}$$

3.2.3 Auflösung der Überlappung des Fehlerintervalls bei Übergängen

Wie bereits in Abschnitt erklärt, wird vorausgesetzt, dass beim Lösen der Differentialgleichung die jeweiligen Lösungen in allen Zeitschritten durch einen maximalen Fehler $err_{max} = \epsilon$ begrenzt sind. Diese Voraussetzung bringt den Vorteil mit sich, dass die Fehlerberechnung nicht als weiterer Schritt wie in [Nie13] gemacht werden muss, sondern einmalig ein festgelegter Fehlerwert vorausgesetzt werden kann und dann zu jedem Zeitpunkt eine Überabschätzung des errechneten Wertes mit err_{max} gemacht werden kann.

In [Nie13] wurde gezeigt, dass ein Modell in eine diskretisierte Form eines Differentialgleichungssystems integriert werden kann, wenn zu einem Zeitpunkt t_i für einen gelösten Wert $x_j(t_i)$ einer Differentialgleichung der entstandene Fehler err_{max} in ein Intervall $[x_j(t_i)] = [x_j(t_i) - err_{max}, x_j(t_i) + err_{max}]$ eingeschlossen wird. Dies wirkt sich auf die im vorherigen Abschnitt beschriebene Auswertung der Übergangsbedingungen aus, da alle Werte der gelösten Differentialgleichung als Intervall betrachtet werden müssen. So könnte es dazu kommen, dass der Vergleich $[x_j(t_i)] < a$ mit $a \in \mathbb{R}$ sich mit dem negierten Vergleich $\neg([x_j(t_i)] < a) = [x_j(t_i)] \geq a$ überschneidet.

Dies führt dann dazu, dass ein Algorithmus zur Anpassung der Zustandsübergänge eingesetzt werden muss wie er in Abschnitt 2.9 beschrieben ist. Im Gegensatz zum Lösen der Differentialgleichung zur Zeit des Model Checking ist es möglich, wie im vorherigen Abschnitt angewendet, die Anpassung der Vergleiche vor dem Modelchecking durchzuführen, weil bereits vor dem Modelchecking bekannt ist, ob sich zwei Vergleiche überschneiden oder nicht. Folglich ist eine Anpassung der Übergänge durch Anpassung der Übergänge oder Neuberechnung des Systems nur notwendig, wenn sichergestellt ist, dass sich zwei Vergleiche überlappen.

Im Folgenden ist in Algorithmus 4 ein Algorithmus beschrieben, der kritische Übergänge im Modell erkennt und anschließend behandelt.

Im ersten Teil wird der Funktion $getCriticalTransitions$ die Aufgabe zugewiesen, alle Transitionen im Modell zu finden, bei denen sich mindestens ein Vergleich nach Einsetzen der Lösungen mit Fehlerintervall überschneidet.

Der zweite Teil ist eine Implementierung der im Abschnitt 2.9 beschriebenen Möglichkeit sich überschneidende Transitionen durch den Einbau von Indeterminismen.

3.2.4 Erkennen von sich überschneidenden Übergangsbedingungen

In diesem Abschnitt wird die Funktion $getCriticalTransitions$ aus dem Algorithmus 4 behandelt. Das Ziel ist es für eine Liste von Übergängen für alle Vergleiche

Algorithm 4 Funktion *handleErrorInTransitionConditions* zur Transformation eines Modell mit aufgelösten Differentialgleichungen

input : The Model H which has to transformed, A Set of k Transitions T, a solved ODE System M
output : The transformed Model H'

```
31 //PART 1
32 C ← getCriticalTransitions(T, M)
33 //PART 2
34 if sizeOf(C) > 1 then
35     for ∀t_i, t_i ∈ C do
36         T ← T \ t_i
37         handleCriticalTransition(t_i)
38         T ← T ∪ t_i
```

einer Übergangsbedingung zu prüfen, ob für ein t_j und den zugehörigen Wert einer Wert der Differentialgleichung $x_i(t_j)$ der Funktionswert innerhalb des Fehlerintervalls $err_{intervall} = [x_i(t_j) - err_{max}, x_i(t_j) + err_{max}]$ liegt.

Algorithm 5 Funktion *getCriticalTransitions* zum Erkennen von sich überschneidenden Übergangsbedingungen

input : A Set of k Transitions T_1, a solved ODE System M
output : A Set Of Transitions T_2

```
39 for i ← 0 to k, t_i ∈ T_1 do
40     C ← getAllComparisons(t_i)
41     for j ← 0 to l, c_j ∈ C do
42         if c_i has ode identifier then
43             if isCritical(c_i, M) then
44                 T_2 ← T_2 ∪ t_i ;
```

In Algorithmus 5 wird vorbereitend über alle Übergänge und anschließend über alle Vergleiche der Übergänge iteriert. Anschließend wird innerhalb der Vergleiche getestet, ob der Vergleich eine Variable enthält, die einer Differentialgleichung zugeordnet ist. Wenn dieser Fall ist, wird anschließend mit der Funktion *isCritical* (Algorithmus 6) getestet, ob dieser Vergleich „kritisch“ ist. Wenn dieser Vergleich kritisch ist, ist auch der gesamte Übergang kritisch und wird in eine Liste der kritischen Übergänge hinzugefügt.

Die Funktion *isCritical* aus Algorithmus 6 hat als Eingangsparameter den aktuell behandelten Vergleich und das gelöste System von Differentialgleichungen. Es werden nun alle Zeitschritte t_0 bis t_n betrachtet und für jeden Zeitschritt werden die Werte für alle Variablen x_i einer Differentialgleichung in diesem Vergleich gesetzt. Unter Berücksichtigung des Fehlerintervalls wird nun mithilfe der Intervallarithmetik das Maximum oder Minimum der linken bzw. rechten Seite des Vergleichs bestimmt.

Anschließend wird in der Funktion *evaluateComparisons* (Algorithmus 7) angewendet. Diese vergleicht Maximum mit Minimum der rechten bzw. linken Seite des Vergleichs. Sofern diese Ergebnisse unterschiedlich sind mit dem ursprünglichen Vergleich, kann aufgrund des Fehlers keine Aussage mehr getroffen werden, ob der Vergleich wahr oder falsch ist.

Algorithm 6 Funktion *isCritical* zum Testen ob alle Variablen innerhalb des Fehlerintervalls liegen

```
input  : Comparison c, M
output : boolean
45 for ∀t_i do
46     sup_left ← left(c)
47     replaceIdentifierWithSolutionSup(sup_left, M, t_i, err_max)
48     inf_left ← c
49     replaceIdentifierWithSolutionInf(inf_left, M, t_i, err_max)
50     sup_right ← right(c)
51     replaceIdentifierWithSolutionSup(sup_right, M, t_i, err_max)
52     inf_right ← right(c)
53     replaceIdentifierWithSolutionInf(inf_right, M, t_i, err_max)
54     if ¬evaluateComparisons(inf_left, sup_left, inf_right, sup_right, c) then
55         return false
56 return true;
```

Algorithm 7 Funktion *evaluateComparisons* zur Auswertung von Vergleichen

```
input  : Expressions: compSupLeft, compInfLeft, compSupRight, comInfRight
input  : Comparison: comparison
output : boolean
57 if compare (compSupLeft, compInfRight, getComparator(comparison) != evaluate(comparison) then
58     return false;
59 if compare(compInfLeft, compSupRight, getComparator(comparison) != evaluate(comparison)) then
60     return false;
61 return true;
```

3.2.5 Zusammenfassung der Transformationsprozesse

Als Abschluss dieses Abschnitts ist der Transformationsalgorithmus in kompletten Form in Algorithmus 8 dargestellt.

Der Algorithmus besteht im Wesentlichen aus den Komponenten aus Abbildung 3.2. Bis auf die im ersten Teil benutzten Funktionen zur Initialisierung des Systems sind alle weiteren benutzten Funktionen wie *handleErrorInTransitionConditions* in den vorherigen Abschnitten eingeführt worden. Die Funktionen im ersten Teil entsprechen dem Auslesen, der in Abschnitt 3.1 eingeführten Eigenschaften des Modells.

3.3 Behandlung eines parametrisierten Modells

Bisher wurden nur Modelle betrachtet, die über Differentialgleichungen $\dot{x} = f(x,t)$ deklariert wurden und nach bestimmten Zeitpunkten Parameter verändern. Im Folgenden soll ein Weg gefunden werden, um Differentialgleichungen mit Parametern über $\dot{x} = f(x,t,p)$ und den Parametervektor

$$p = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

zu betrachten.

3.3.1 Kennzeichnung von Parametern im Ausgangsmodell

Als erster Schritt ist es notwendig im Modell den Parametervektor p zu definieren. Der Parametervektor kennzeichnet sich durch einen gegebenen Anfangsvektor p_0 aus,

Algorithm 8 Pseudocode der Transformation in ein Modell mit aufgelösten Differentialgleichungen

```
input  : The Model H
output : The transformed Model H'
62 //PART 1: INITIAL TRANSFORMATION
63 ẋ ← getODESystem(H)
64 TIME ← getTimeSystem(H)
65 deleteODEDefintions (H, ẋ)
66 transformTimeSystem(H, TIME)
67 //PART 2: SOLVING SYSTEM
68 M ← solveODESystem (ẋ, TIME)
69 //PART 3
70 T ← getTransitions(H);
71 //PART 4
72 handleErrorInTransitionConditions (H, M, T)
73 //PART 4: HANDLE ALL TRANSITIONS
74 replaceComparisonsWithTimeSteps (H, M, T)
75 return H
```

der jedem Parameter einen Anfangswert zum Zeitpunkt t_0 zuweist. Weiterhin ist es nötig im Kontext des Model Checking einen diskreten Wertebereich zu definieren, um ein endliches und diskretes Modell zu erstellen.

3.3.2 Auflösung von Parametern in den Differentialgleichungen

Im Folgenden wird der Parametervektor p als ein Vektor p_t betrachtet. Für jedes $t \in \{t_0, t_1, \ldots, t_n\}$ existiert eine zugehörige Parameterkombination. Für jedes p_t muss zum betrachteten Zeitpunkt t_i der Parameter der Differentialgleichung ersetzt werden und die Differentialgleichung beginnend ab t_i mit den Anfangswerten

$$x(t_i) = \begin{pmatrix} x_1(t_i) \\ x_2(t_i) \\ \vdots \\ x_i(t_i) \\ \vdots \\ x_n(t_i) \end{pmatrix} \tag{3.5}$$

betrachtet werden. Für das Beispielsystem

$$\dot{x} = \begin{pmatrix} \dot{x}_1 = p_2 \\ \dot{x}_2 = x_1 + p_1 \end{pmatrix} \tag{3.6}$$

müssen demzufolgen die Parameter p_1 und p_2 für t_i mit den Parameterbelegungen ersetzt werden. Sind $p_1 = 1$ und $p_2 = 2$ ergibt sich daraus zum betrachteten Zeitpunkt t_i das System:

$$\dot{x} = \begin{pmatrix} \dot{x}_1 = 2 \\ \dot{x}_2 = x_1 + 1 \end{pmatrix} \tag{3.7}$$

Sofern $p_{t0} = p_{t1} = \cdots = p_{tn}$, also keine Änderung der Parameter über das betrachtete Zeitintervall stattgefunden hat, ist das modellierte Verhalten gleich zu

dem Verhalten das in den bisherigen Abschnitten betrachtet wurde. Eine Änderung der Konstellation ergibt sich, wenn für einen Zeitpunkt t_i gilt $p_{ti} \neq p_{ti+1}$, also ein Wechsel der Parameter stattgefunden hat. In diesem Fall ist es notwendig den Parametervektor p_{ti+1} zu bestimmen.

Da nur Erreichbarkeitsprobleme betrachtet werden, ist es möglich den Zustandsvektor mithilfe eines Model Checker zu bestimmen, der ein Gegenbeispiel zu einer gegebenen Spezifikation bestimmen kann. Um dies zu erreichen muss, die Erreichbarkeitsspezifikation angepasst werden auf alle Zustände, für sich der initiale Wert genau eines Parameters verändert. Für eine gegebene Beispielspezifikation

3.3.3 Auswertung der Ergebnisse des Model Checker

Aus dem entstehenden Gegenbeispiel lässt sich ablesen, welche Wertbelegungen alle Parameter im Modell zu einem bestimmten Zeitpunkt besitzen. Da die Zeit ebenso also Parameter modelliert ist, lässt sich auch der Zeitpunkt des Parameterwechsels genau bestimmen. Sollte ein Wechsel eines Parameters zum Zeitpunkt t_i einer Differentialgleichung auftreten, muss zu diesem Zeitpunkt eine Neuberechnung des Modells stattfinden. Da die Lösung des Modells unter der vorherigen Parameterkonstellation zu diesem Zeitpunkt bekannt ist, lässt sich der neue Anfangswertvektor über die vorher bestimmte Lösung M bestimmen:

$$x_0(t_i) = \begin{pmatrix} M_{1i} \\ M_{2i} \\ \vdots \\ M_{ni} \end{pmatrix} \tag{3.8}$$

Anhand der Lösung des zum Zeitpunkt t_i entstandenen Differentialgleichungssystems kann das in den vorherigen Abschnitten beschriebene transformierte Modell entstehen.

3.3.4 Behandlung von indeterministischen Verhalten

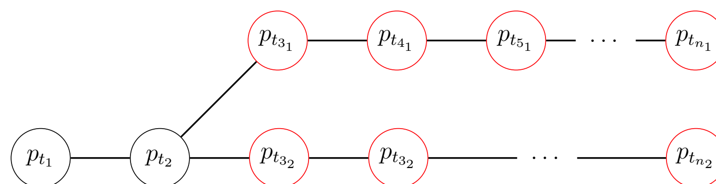

Abbildung 3.4: Nichtdeterministisches Verhalten der Parameter im Verlauf des Model Checking

Bis jetzt wurde noch nicht das Auftreten von indeterministischen Übergängen im Modell betrachtet. Der bisherige Ansatz beruhte darauf, dass sofern sich ein Parameter ändert, ein Gegenbeispiel vom Model Checker zurückgegeben wird und aus diesem die neue Parameterkonstellation abgelesen wird. Sollte aber, wie in Abbildung 3.4 dargestellt, für eine Parameterkonstellation ein Indeterminismus modelliert sein (in der Abbildung für die Parameterkonstellation p_{t2}), dann verfolgt der Modelchecker einen der beiden entstehenden Zweige.

Wenn nun für die Parameterkonstellation p_{t5} ein Parameterwechsel stattfindet, wird nach dem bisherigen Verfahren anhand des entstandenen Gegenbeispiels die

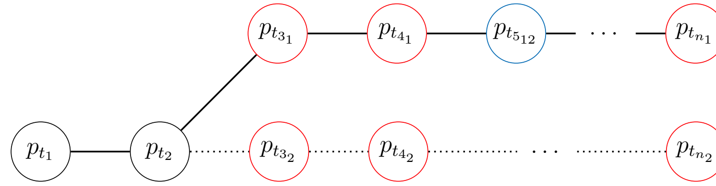

Abbildung 3.5: Wechsel eines Parameters bei nichtdeterministischen Verhalten im Verlauf des Model Checking

Differentialgleichung mit den gewechselten Parametern neu berechnet. Anschließend wird das Modell ein weiteres Mal gecheckt. Dabei wird die Information verworfen, dass indetermistisches Verhalten bei der Parameterkonstellation p_{t2} vorlag. Dementsprechend wird der zweite entstehende Zweig nicht betrachtet und verworfen.

Dieser Zusammenhang ist in Abbildung 3.5 dargestellt. Der blau eingefärbte Knoten p_{t5} signalisiert den Parameterwechsel. Dass ein weiterer Zweig noch zu prüfen ist, ist ausgehend von der Parameterkonstellation p_{t5} nicht ersichtlich.

Diese Situation kann mithilfe von Backtracking (Algorithmus 9) gelöst werden. Bei Backtracking wird solange in vergangenen Knoten gesucht, bis eine Lösung gefunden wird. Eine Lösung in diesem Fall ist dadurch definiert, dass ein Knoten zwei Kinder besitzt. Für das eingeführte Beispiel werden bei der Anwendung des Backtrackings (dargestellt in Abbildung 3.5) von p_{t5} die Knoten p_{t4} und p_{t3} geprüft bis letztendlich mit p_{t2} eine Lösung gefunden ist, da der Knoten zwei Kinder besitzt.

Algorithm 9 Backtracking Verfahren

input : Start Node N
output : Node

```
76 while hasParent(N) do
77     parent ← getParent(N)
78     children ← getChildren(parent)
79     if sizeOf(children) > 1 then
80         return parent
81 return ∅
```

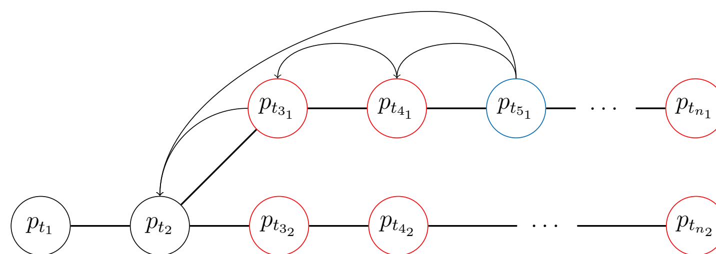

Abbildung 3.6: Backtracking zum Start des Nichtdeterminismus

Um Backtracking anwenden zu können, werden ausgehend von dem Knoten, in dem ein Parameterwechsel stattfindet, Informationen zu vergangenen Knotenpunkten benötigt. Diese Informationen müssen aus dem Modelchecker extrahiert werden, um das Backtracking anwenden zu können.

4. Implementierung des Transformators

Das folgende Kapitel beschreibt eine Umsetzung des im vorherigen Abschnitts vorgestellten Transformationsalgorithmus. Zuerst wird beschrieben, wie in SAML ein System von Differentialgleichungen modelliert wird. Im Anschluss geht das Kapitel auf die Implementierung der Schnittstelle zu dem hier benutzten Lösungsprogramm für Differentialgleichungen Scilab aufgebaut ein und wie die Ergebnisse intern gespeichert werden. Zu Abschluss soll darauf eingegangen werden, wie die Implementierung mit Indeterminismen im Modell umgeht.

4.1 Aufbau eines hybriden Automat in SAML

In diesem Abschnitt wird zunächst beschrieben, wie in SAML die Zeit als Komponente modelliert werden kann und welchen Einfluss diese auf ein SAML-Modell hat. Der zweite Teil diskutiert zwei Wege, um Differentialgleichungen in SAML modellieren zu können.

4.1.1 Bilden des Zeitsystems in SAML

Um die Modellierung der Zeit in diesem Rahmen auf das Wesentliche zu reduzieren, wurde im Rahmen der Arbeit in SAML eine globale Definition der Zeit eingeführt. Dies ist in Quelltext 4.1 dargestellt.

Quelltext 4.1: Modellierung der globalen Zeit

```
t: [[START..END], deltaTime];
component x
        ...
endcomponent
```

Diese Definition wird genau einmal global in einem SAML-Modell erlaubt, um mehrdeutige Definitionen der Zeit zu vermeiden. Im Umkehrschluss bedeutet dies, dass alle im Modell definierten Differentialgleichungen dieser globalen Definition der Zeit folgen. Es wird nicht näher festgelegt, in welcher Einheit die Zeit modelliert

werden muss. Dies steht dem Modellierenden frei. Sofern er jedoch eine Einheit für die Zeit gefunden hat, muss er im weiteren Modellverlauf dieser Modellierung folgen.

Im Transformationsprozess wird diese Deklaration in die Zeitkomponente aus Quelltext 5.1 überführt.

4.1.2 Modellierung homogener Differentialgleichungen

Für eine Modellierung in SAML bietet sich an, die Deklaration von Differentialgleichungen ähnlich zu der von Zustandsparametern zu halten. In dieser Arbeit wird eine Differentialgleichung $\dot{x}(t) = a$ mit $a \in \mathbb{R}$ als Kommentar $/*@ODE : x' = a$ einer Zustandsvariablendeklaration deklariert.

Ein System von Differentialgleichungen $\dot{x}(t, x)$ entspricht allen Differentialgleichungen innerhalb des Modells. Für das Beispielsystem

$$\dot{x} = \begin{pmatrix} \dot{x}_1 = 2 \\ \dot{x}_2 = x_1 + 1 \end{pmatrix} \tag{4.1}$$

mit den Anfangswerten $x_0 = (0, 1)^T$ ergibt sich die in Quelltext 4.2 dargestellte Deklaration in SAML.

Quelltext 4.2: Modellierung von Differentialgleichungen in SAML

```
/*@ODE:x1' = 2*/
x1: [[START..END] init 0;
/*@ODE:x2' = x1 + 1*/
x2: [[START..END] init 1;
```

Diese Deklaration hat im Rahmen dieser Arbeit allein praktische Zwecke, da ein bekanntes Sprachkonzept verwendet wird und ergänzt wird. In dieser Arbeit spielt es keine Rolle, in welchem Intervall die Differentialgleichung definiert sind, weshalb dieses Sprachelement für Differentialgleichungen überflüssig wäre, aber trotzdem aufgrund der generellen Verwendung von Deklarationen in SAML beibehalten wird.

4.2 Umsetzung der Transformationsprozesse

Dieser Abschnitt beschreibt, in welcher Umgebung die Transformationsprozesse innerhalb von VECS umgesetzt sind. Auf Abbildung 4.1 sind die Abläufe der im Transformationsprozess beteiligten Klassen als Sequenzdiagramm dargestellt.

Die Transformation findet hauptsächlich in der Klasse ODETranslator statt. Diese Klasse baut auf einer Struktur von Klasse auf, die dazu dienen, ein gegebenes Model in SAML in ein umgewandeltes Model in SAML zu transformieren.

Jedes Element wie eine Differentialgleichung oder ein Zustandsübergang wird in dieser Struktur ein Transformationsergebnis zugewiesen. Die Klasse ODETranslator erbt von dieser Transformationsstruktur und überlädt die Funktionen, die die gewünschten Elemente wie die Deklaration der Zeit oder Definition der Differentialgleichungen implementieren. So wird das Ausgangsmodell in SAML übersetzt, während einige gewünschte Elemente transformiert werden.

Ausgehend vom ODETranslator wird zuerst aus dem Model mithilfe eines „System Builders“ das System von Differentialgleichungen ausgelesen. Anschließend wird mit dem „Time Component Builder“ das Zeitsystem ausgelesen.

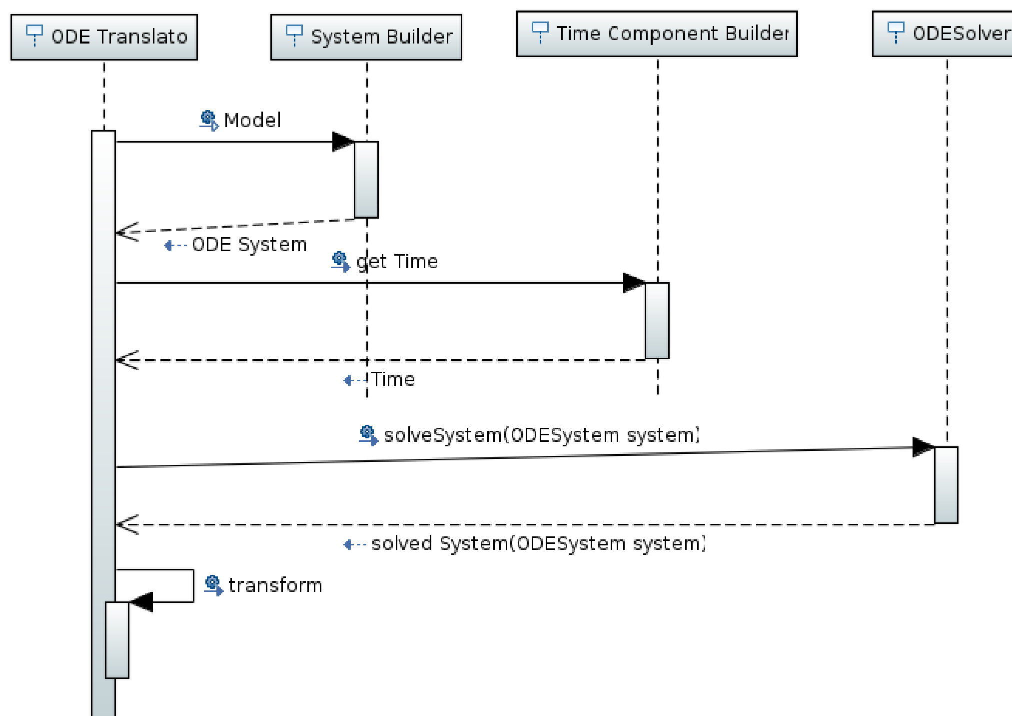

Abbildung 4.1: Sequenzdiagramm der Transformationsschritte

Das so ausgelesene System von Differentialgleichungen und das Zeitsystem kann nun an einen ODE-Solver übergeben werden und gelöst werden. Abschließend werden basierend auf gelösten System die weiteren Transformationen durchgeführt.

4.3 Schnittstelle zwischen ODE-Solver und Transformer

Der folgende Abschnitt beschreibt, wie das Lösungsprogramm für Differentialgleichungen umgesetzt wurde und welche Datenstrukturen verwendet werden, um die erzielte Lösung für den Transformer verwendbar zu machen. Ein Ziel der Umsetzung ist es, dass sie nicht ausschließlich für ein Lösungsprogramm für Differentialgleichungen anwendbar ist, sondern auch weitere Programme so angebunden werden können.

4.3.1 Scilab

Der verwendete Prototyp verwendet Scilab als Lösungsprogramm für Differentialgleichungen verwendet. Scilab ist in der Anwendung sehr ähnlich zu Matlab, das ebenso als Lösungsprogramm verwendet werden könnte[SR97]. Scilab bietet ein Interface zu Java an [Enta]. Über diese Schnittstelle kann Scilab-Code ausgeführt werden und auf die Ergebnisse zurückgegriffen werden.

In Scilab gibt es die Möglichkeit mit der Funktion *ode* ein System von Differentialgleichungen für einen gegebenen Zeitraum zu lösen [Entb].

Im Folgenden ist in Quelltext 4.3 ein Beispiel für den verwendeten Scilab-Code gegeben, dessen Struktur [CCN06] entnommen ist.

In dem Beispiel wird im Zeitraum von 0 bis 90 mit $\Delta t = 90$ folgendes Anfangswertproblem gelöst:

Quelltext 4.3: Modellierung und Lösung eines Systems von Differentialgleichungen in Scilab

```
function dx=f(t, x)
   dx(1) = 1;
   dx(2) = x(1);
   dx(3) = 1;
   dx(4) = x(3);
endfunction;
x0= [33.0;300.0;33.0;0.0];
t0=0;
t=0:0.1:90;
err_relative = 0;
err_absolute = 0.00000000001;
x = ode(x0,t0,t,err_relative,err_absolute,f);
```

$$(x_1, x_2, \dot{x}_3; x_4)^T(t) = \begin{pmatrix} 1 \\ x_1 \\ 1 \\ x_3 \end{pmatrix} \tag{4.2}$$

mit den Anfangswerten $x_1(0) = 33$, $x_1(0) = 300$, $x_1(0) = 33$ und $x_4(0) = 0$.

Scilab entspricht nicht ganz dem gewünschten Interface wie es in Abbildung 3.3 dargestellt und im Abschnitt Abschnitt 3.2.1 beschrieben wurde. Das liegt daran, dass Scilab keine Fehlerberechnung durchführt, sondern in jedem Diskretisierungsschritt einen Fehler $est(i)$ abschätzt. Über eine absolute und relative Toleranzgrenze kann geregt werden, wie genau Scilab die numerische Lösung ermittelt [Sal]. Folgende Formel muss für einen Diskretisierungsschritt erfüllt sein:

$$est(i) \leq rtol(i)|x(i)| + atol(i)$$

In dieser Arbeit wird eine absolute Toleranzgrenze von 10^{-11} verwendet. Es ist wichtig zu beachten, dass ein berechneter Wert x_i nicht zwingend im Intervall $[x_i - 10^{-11} \leq x_i \leq 10^{-11}]$ liegt. Für diese Arbeit wird im Folgenden angenommen, dass dies jedoch der Fall ist, um ein Lösungsprogramm wie Scilab mit den Standardfunktion zum Lösen von Differentialgleichungen zu benutzen.

Dadurch kann der Prototyp keine feste Aussage treffen, ob für manche Fälle das Modell unsicher ist, da der Fehler für die gelöste Differentialgleichung auch außerhalb des verwendeten Fehlerintevalls liegen könnte. Dies ist jedoch aufgrund der sehr kleinen Toleranzgrenze für die meisten Modelle sehr unwahrscheinlich.

4.3.2 Schnittstellen ODE Solver und ODE Solution

Um den Transformer nicht abhängig von dem gewählten Lösungsprogramms wie Scilab zu machen, wurde für diese Implementierung ein Interface „ODESolver“ erstellt. Dieses stellt die Methode *solve* zur Verfügung, mit der ein gegebenes System von Differentialgleichungen gelöst werden kann. Im Fall von Scilab wird das Interface durch die Klasse „ScilabODESolver“ implementiert.

Das gesamte implementierte Klassensystem ist in Abbildung 4.2 als Klassendiagramm dargestellt.

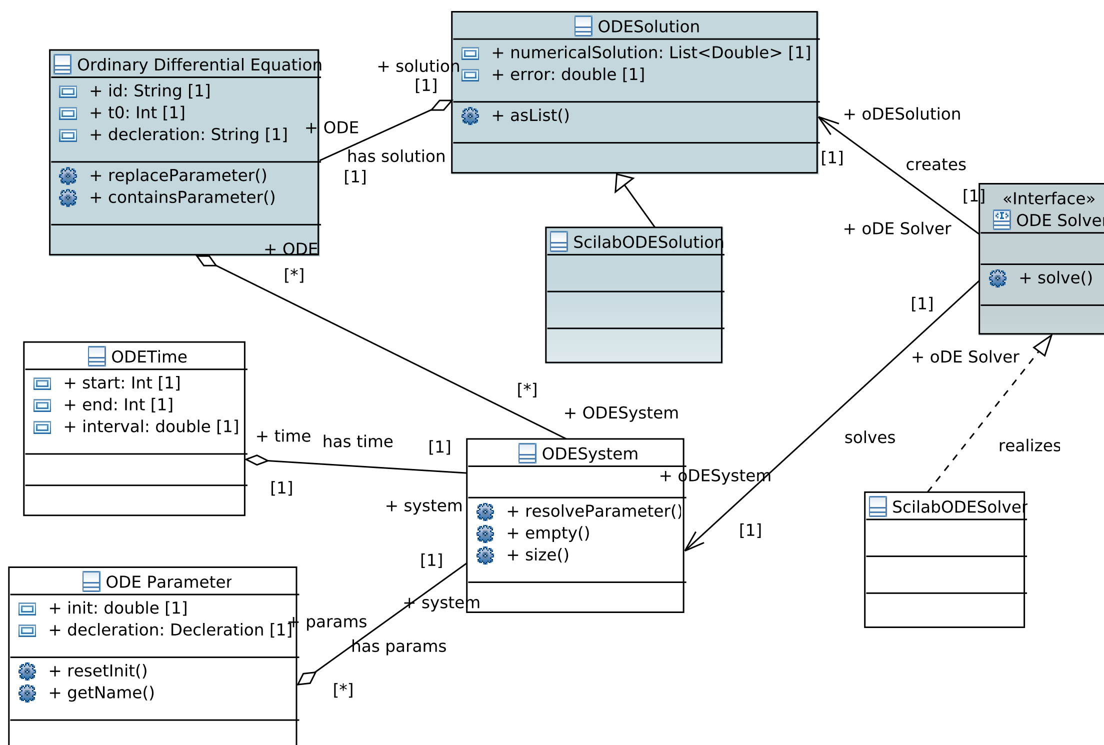

Abbildung 4.2: Datenstrukturen zur Anwendung und Speicherung der Differentialgleichungen als Klassendiagramm

Die zentrale Klasse ist die Klasse „ODESystem“, die ein System von Differentialgleichungen implementiert. Außerdem besitzt diese Klasse eine Menge von im Modell verwendeten Parametern, die über die Funktion *resolveParameter* durch den gesetzten Initialwert im SAML-Modell in den zugehörigen Differentialgleichungen aufgelöst werden können.

Sofern die Methode *solve* aus der „ODESolver“-Klasse aufgerufen wird, trägt diese Klasse eine Instanz der Klasse „ODESolution“ in jede einzelne Differentialgleichung ein. Diese Lösung besteht aus einer Liste von Zahlenwerten, die die Länge n (Anzahl der Zeitschritte) besitzt. Die Klasse kann durch eine Klasse für das jeweils verwendete Lösungsprogramm verwendet werden, um z.B. bei der Konstruktion einer Instanz das Ergebnis des Lösungsprogramms in das Listenformat der Hauptklasse zu transformieren.

4.4 Kommunikation von Modelchecker und VECS

In der bisher vorgestellten Implementierung wurden parametrisierte Differentialgleichungssysteme, wie sie in Abschnitt 3.3 beschrieben wurden, nicht näher betrachtet. Sofern das Modell keine Parameter beinhaltet, kann das Ergebnis des Transformer an jeden symbolischen Model Checker angebunden werden, für den ein Algorithmus zur Übersetzung aus SAML existiert.

In dieser Implementierung wird ein Parameter im Modell mit dem Kommentar

/@*ODE-Parameter*/gekennzeichnet. Dies ist beispielhaft in Quelltext 4.4 umgesetzt

Quelltext 4.4: Kennzeichnung von Parametern in einem SAML-Modell

```
/@*ODE-Parameter*/
param: [[START..END]] init INIT;
```

Diese Kennzeichnung ist nicht zwingend notwendig, aber sie reduziert die Möglichkeiten, dass eine Differentialgleichung durch einen Parameter beeinflusst werden kann. Außerdem vereinfacht eine solche Definition vorerst den Programmieraufwand, da man anhand der Kennzeichnung als Parameter einer Differentialgleichung alle derartigen Parameter aus dem Modell herauslesen kann anstatt alle Differentialgleichungen auf Parameter und deren Existenz zu überprüfen.

In zukünftigen Umsetzungen wäre es jedoch vorzuziehen, dass die Parameter über die Differentialgleichungen mitdefiniert werden und innerhalb der IDE eine Überprüfung der Parameter auf deren Existenz stattfindet. Dies hat den Vorteil, dass es keinen Kommentar benötigt, der den Parameter als Parameter von Differentialgleichungen kennzeichnet und in der IDE dargestellt werden kann, ob und an welcher Stelle der Parameter sich im Modell befindet.

Durch die eingeführten Parameter in Differentialgleichungen, muss es möglich sein, auf die Belegung der Parameter im Prozess des Model Checking zurückzugreifen, um bei einer Änderung der Parameter eine erneute Transformation des Modells durchführen zu können.

In Abbildung 4.3 ist als Sequenzdiagramm der Ablauf dargestellt der Kommunikation zwischen dem Transformer und dem Modelchecking-Prozess dargestellt.

Zuerst wird, wie in den vorherigen Abschnitten beschrieben, das Model initialisiert und transformiert.

Für den Fall, dass im Modell ein oder mehrere Indeterminismen vorhanden sind, muss wie in Abschnitt 3.3.4 wurde beschrieben, jeder Indeterminismus im Modell betrachtet werden. Für den Fall, dass ein Zweig mit keinem Gegenbeispiel endet, wird das Backtrackingverfahren angewendet, um weitere Pfade zu prüfen. Das Problem besteht nun darin, dass der Modelchecker den Indeterminismus auswertet und somit a priori nicht festzumachen ist, in welchem Zustand ein Indeterminismus vorgefunden wurde.

Um dies zu vermeiden wird im Ablauf, eine Detektion von Indeterminismen durchgeführt. Sofern ein Indeterminismus gefunden wurde, wird für jede mögliche neue Belegung ein neues Modell erstellt, das von dem Model Checker gecheckt wird.

Zu beachten ist, dass in der Implementierung der letzte Teil nicht vollständig implementiert ist. Im Fall dass Indeterminismen gefunden werden, werden ausschließlich die entstandenen Modelle zurückgegeben. Für diese müsste nun separat ein neuer Prozess des Model Checking angestoßen werden.

In der Implementierung wurde als symbolischer Modelchecker NuSMV verwendet. Als Anbindung wurde bereits die in der VECS-Umgebung vorhandene Struktur eines Übersetzers und eines Adapters genutzt. Der Prototyp verwendet die in dieser Implementierung Schnittstelle zum symbolischen Modelchecker. Es besteht auch eine Möglichkeit NuSMV mit einem Bounded Model Checking Ansatz einzusetzen.

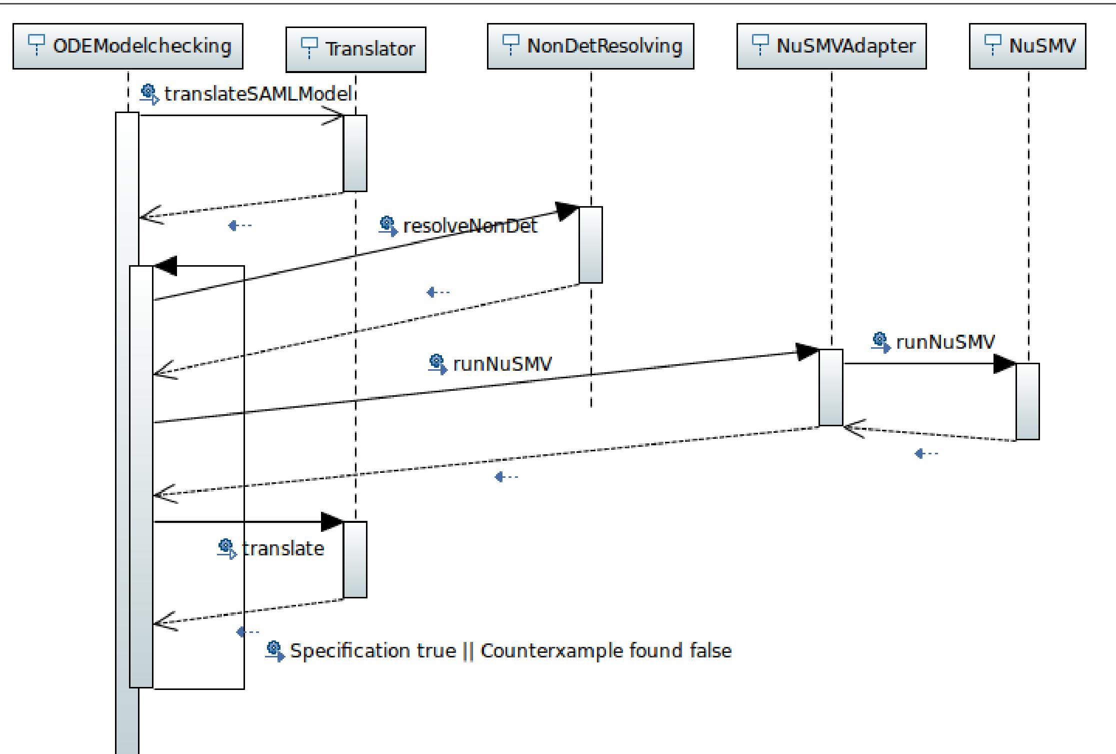

Abbildung 4.3: Sequenzdiagramm der Abläufe in der Kommunikation zwischen Prozessen des Model Checking und der Transformation

Wenn ein Gegenbeispiel für die gegebene Spezifikation gefunden wurde oder alle Pfade abgesucht wurden, endet die Berechnung.

5. Beschreibung des Verfahrens anhand von zwei Fallstudien

In diesem Kapitel werden zwei Fallstudien eingeführt. Mit diesen wird die Funktionsweise des in den vorherigen Abschnitten eingeführten Verfahrens erläutert. Anhand der Fallstudien werden im nachfolgenden Kapitel Experimente durchgeführt, um die Eigenschaften im Vergleich zu anderen Ansätzen herauszustellen.

5.1 Punktförmige Zugbeeinflussung (Indusi)

Als erste Fallstudie wird in dieser Arbeit die in [Nie13] eingeführte Fallstudie einer punktförmigen Zugbeeinflussung verwendet. Zuerst wird die Fallstudie näher beschrieben. Im Anschluss wird die Transformation des Modells mit dem in dieser Arbeit vorgestellten Verfahren erläutert.

5.1.1 Beschreibung

Eine punktförmige Zugbeeinflussung ist in der Lage in besonderen Fällen, Einfluss auf einen Zug auszuüben. Ein Anwendungsfall könnte z.B. eine Kurve auf einer Strecke sein, die nur mit einer bestimmten Maximalgeschwindigkeit durchfahren werden darf. Sollte der Zugführer den Zug bis zu einem bestimmten Ort vor der Kurve, nicht die Geschwindigkeit angepasst haben, so kann das System die Geschwindigkeit des Zugs nach diesem Zeitpunkt reduzieren.

Abbildung 5.1: Schematische Darstellung des automatischen Bremsverhaltens bei einer punktförmigen Zugbeeinflussung (aus: [Nie13])

In Abbildung 5.1 ist dieser beschriebene Beispielfall schematisch dargestellt. Ein Zug fährt in Abhängigkeit einer Zeit t mit einer Geschwindigkeit $vel(t)$ und besitzt eine Position $pos(t)$. Zu Beginn erhält der Zug ein Signal, seine Geschwindigkeit zu reduzieren. Wenn der Zug seine Geschwindigkeit zum Abschnitt mit reduzierter Geschwindigkeit reduziert hat, wird ein Alarmsignal gesetzt. Die Fragestellung an dieses Modell ist nun, ob das Signal für ein bestimmtes Anfangswertproblem gesetzt wird.

Für diese Fallstudie hat er der Zug eine Geschwindigkeit von $50m/s$. Die Veränderung der Geschwindigkeit ist als Differentialgleichung modelliert, sodass sich folgendes Anfangswertproblem ergibt:

$$\dot{vel}(t) = t - 10 \quad vel(t = 0) = 50[m/s]$$

Für die Position ergibt sich folgendes Anfangswertproblem:

$$\dot{vel}(t) = vel(t) \quad pos(t = 0) = 0[m]$$

Das Beispiel lässt sich nun als hybrider Automat modellieren (siehe Abbildung 5.2).

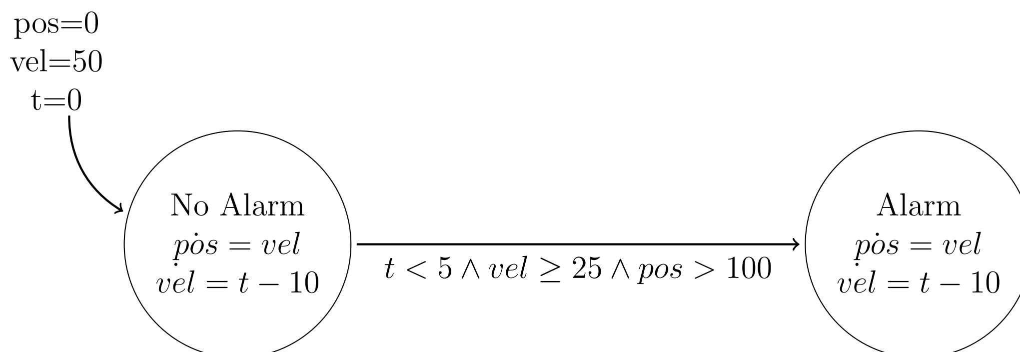

Abbildung 5.2: Darstellung des Beispiels der punktförmigen Zugbeeinflussung als hybrider Automat

Der Automat besitzt mit „No Alarm“ und „Alarm“zwei Zustände. Der Automat beginnt im Zustand „No Alarm“ und wechselt in „Alarm“, wenn eine Sicherheitseigenschaft verletzt ist. Hier ist dies der Fall, dass der Zug nach 5 Sekunden und 100 Metern noch eine Geschwindigkeit von mehr als 25 Metern pro Sekunde besitzt. Die Spezifikation an diesen Automaten ist das Erreichbarkeitsproblem, ob der Automat je den Zustand „Alarm“ erreicht.

5.1.2 Ablauf der Transformationen

Im Anhang ist in Abschnitt A.1 eine Umsetzung des Automat in SAML angegeben. Die Umsetzung folgt der in Abschnitt 3.1 und Abschnitt 4.1 erklärten Methodik zur Umsetzung eines hybriden Automat in SAML.

In dem Automaten wurde jedoch verzichtet, den Sachverhalt zu beachten, dass der Übergang in den Zustand „Alarm“ nur erfolgt, sofern nicht mehr als fünf Sekunden vergangen sind, da die Zeit als Übergangsbedingung nicht implementiert wurde. Für dieses Beispiel ist diese Übergangsbedingung nicht von herausragender Bedeutung, da der Alarm in dem Beispiel unterhalb der fünf Sekunden ausgelöst wird. Somit ist dieser Zusammenhang mit dem späteren Vergleich nicht von großer Bedeutung.

Im Folgenden wird Algorithmus 8 angewendet. Zuerst wird das System von Differentialgleichungen extrahiert. Dieses wurde im vorherigen Abschnitt bereits beschrieben. Als nächstes wird aus dem Model das zeitliche Verhalten extrahiert und anschließend in eine Zeitkomponente transformiert. Die entstandene zeitliche Komponente hat für die betrachteten sechs Sekunden mit der Startzeit $t_0 = 0$ und der Schlusszeit $t_n = 60$ für den gegebenen Automaten die folgende Form:

Quelltext 5.1: Transformierte Zeitkomponente

```
component CLOCK
  TIME : [0..60] init 0;
  TIME + 1 < 60 -> TIME' = TIME + 1;
  TIME + 1 >= 60 -> TIME' = TIME;
endcomponent
```

Anschließend wird das System von Differentialgleichungen mithilfe von Scilab gelöst. Daraus ergibt sich nun die folgende Zustandsmatrix M:

$$M = \begin{pmatrix} vel(t_0) & vel(t_1) & \dots & vel(t_i) & \dots & vel(t_{60}) \\ pos(t_0) & pos(t_1) & \dots & pos(t_i) & \dots & pos(t_{60}) \end{pmatrix} \tag{5.1}$$

Der nächste Schritt besteht darin zu prüfen, ob es eine Fehlerüberlappung im Modell gibt. Dafür werden im ersten Schritt die gefundenen Lösungswerte der Differentialgleichungen für alle Zeitpunkte in das Modell eingefügt. Für die Geschwindigkeit $vel(10)$, also die Geschwindigkeit nach einer Sekunde, wird für alle Übergangsbedingungen, in denen die Geschwindigkeit als Variable vorkommt, die Geschwidigkeitsvariable durch den Wert der Geschwindigkeit nach zehn Sekunden ersetzt.

Für die Übergangsbedingung $VEL \geq 25$ aus dem Ursprungsmodell ergibt sich nun folgende ersetzte Übergangsbedingung:

$$vel(10) \geq 25 \rightarrow 41.405 \geq 25$$

Diese Bedingung könnte nun hinsichtlich ihres Wahrheitswertes aufgelöst werden. Vorher findet jedoch die Fehlerbetrachtung statt. Hierfür wird der Fehlerwert auf den Wert für $vel(10)$ addiert oder subtrahiert, sodass folgende Übergangsbedingungen $41.405 + \epsilon \geq 25$ und $41.405 - \epsilon \geq 25$ entstehen.

Sollten beide Bedingungen denselben Wahrheitswert besitzen, hat der entstandene Fehler keine Auswirkung. Wenn die Wahrheitswerte unterschiedlich sind, wird der in Abschnitt 2.9 beschriebene Algorithmus zur syntaktischen Anpassung von Übergängen angewendet. Da das hier - und im gesamten Modell - für den vorgegebenen Lösungsbereich von kleiner 10^{-11} nicht gegeben ist, wird der Algorithmus hier nicht angewendet.

Abschließend werden im Modell die Zustandsübergangsbedingungen so angepasst, dass die zeitliche Abhängigkeit enthalten ist.

Dafür wird die Funktion *getTimedCondition* aus Algorithmus 3 angewendet, sodass für das Beispiel nach einer Sekunde folgende Bedingung entsteht:

$$41.405 \geq 25 \wedge CLOCK.TIME = 10$$

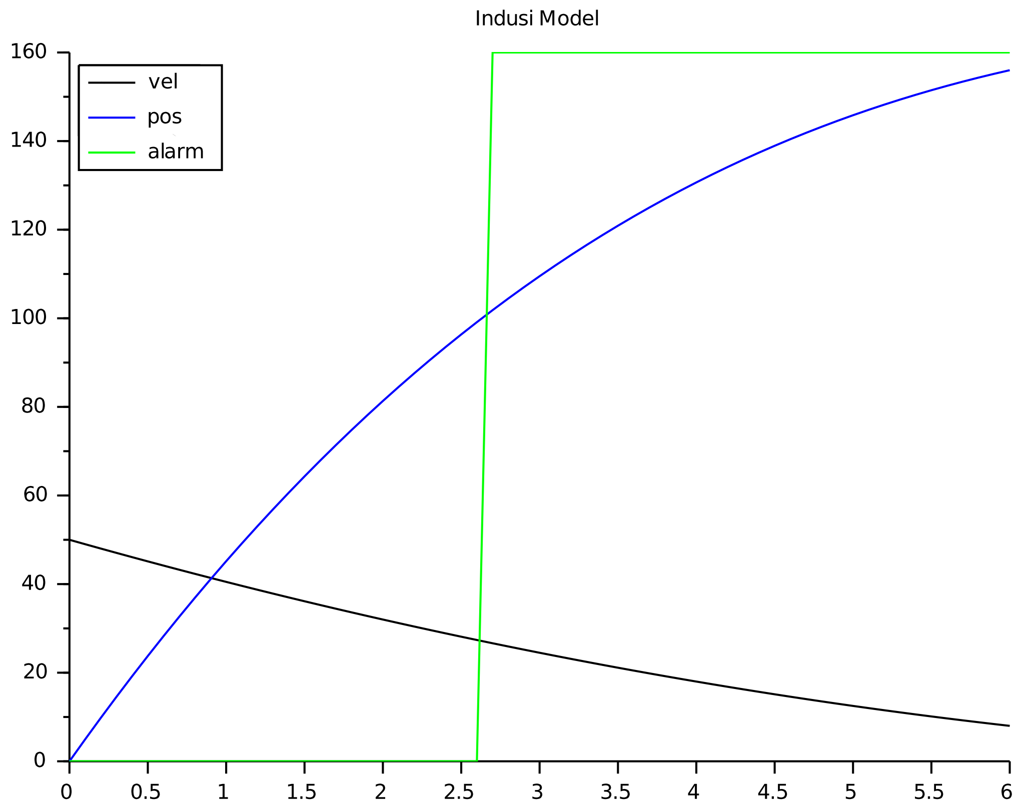

Abbildung 5.3: Darstellung des Verlaufs von $vel(t)$ und $pos(t)$ für das Indusi-Modell

Abschließend können die Zustandsübergangsbedingungen so vereinfacht werden, dass die Vergleiche ausgewertet werden und ausschließlich zeitliche Abhängigkeiten im Modell vorhanden sind. Dieser Schritt wird wieder durch die Übergangsbedingung $VEL \geq 25$ verdeutlicht. Nach dem Einsetzen der Lösung der Differentialgleichung und den zeitlichen Abhängigkeiten entsteht durch Oder-Verknüpfungen (vgl. Algorithmus 3) folgende Übergangsbedingung:

$$
\begin{aligned}
50 \geq 25 \wedge & CLOCK.TIME = 1 \vee \\
49.005 \geq 25 \wedge & CLOCK.TIME = 2 \vee \\
48.002 \geq 25 \wedge & CLOCK.TIME = 3 \vee \\
& \vdots \\
25.205 \geq 25 \wedge & CLOCK.TIME = 30 \\
24.405 \geq 25 \wedge & CLOCK.TIME = 31 \\
& \vdots \\
8.00 \geq 25 \wedge & CLOCK.TIME = 60
\end{aligned}
$$

Nach Vereinfachung dieser Bedingungen ergibt sich eine Bedingung, die nur von der Zeit abhängt:

$$\begin{aligned} CLOCK.TIME &= 1 \vee \\ CLOCK.TIME &= 2 \vee \\ CLOCK.TIME &= 3 \vee \\ &\vdots \\ CLOCK.TIME &= 30 \end{aligned}$$

In Abschnitt A.2 ist das transformierte Modell aufgeführt. Übersetzt und übergibt man dieses Modell einem symbolischen Modelchecker (wie NuSMV) erhält man das Resultat, dass ein Gegenbeispiel gefunden wurde. Für $CLOCK.TIME = 28$ wird der Zustand „Alarm“ erreicht. Dieser Zusammenhang ist in Abbildung 5.3 dargestellt.

Der Plot, der mit einem $\Delta t = 0.1$ berechnet wurde, stellt über die Zeit die Geschwindigkeit und die Position des Zugs dar. Aus dem Plot ist herauszulesen, dass die Bedingungen für das Alarmsignal (größere Geschwindigkeit als $25[m/s]$ und größere Position als 100 Meter) nach ungefähr 2.8 Sekunden wahr werden. Dies ist hier durch die grüne Sprungfunktion gekennzeichnet, die das dann eingetretene Alarmsignal darstellen soll.

5.2 Adaptive Cruise Control

Das zweite Beispiel führt die bereits in Abschnitt 1.1 beschrieben Adaptive Cruise Control (ACC) näher ein. Zuerst wird die Fallstudie beschrieben. Dann wird ein Beispielszenario eingeführt, an dem später Experimente durchgeführt werden sollen. Anhand dieses Szenarios soll die Behandlung von Parametern in einem Modell erläutert werden.

5.2.1 Beschreibung

In dieser Fallstudie werden zwei Fahrzeuge (car_1 und car_2) betrachtet, die jeweils über eine Position in Metern (p_1 für das erste Fahrzeug und p_2 für das zweite Fahrzeug) Geschwindigkeit in Metern pro Sekunde (v_1 und v_2) und eine Beschleunigung in Metern pro Quadratsekunde (a_1 und a_2) verfügen.

In Abbildung 5.4, die [LPN11] entnommen ist, ist das zu betrachtende Szenario dargestellt. Beide Fahrzeuge fahren mit einer bestimmten Geschwindigkeit. Ab einer Position p_{stop} bremst das vorausfahrende Fahrzeug (Fahrzeug 1). Das hinterherfahrende Fahrzeug muss in diesem Fall auf diesen Bremsvorgang reagieren. Sollte diese Regelung nicht funktionieren, könnte es zum Zusammenstoß kommen, was für dieses Szenario die Spezifikation darstellt.

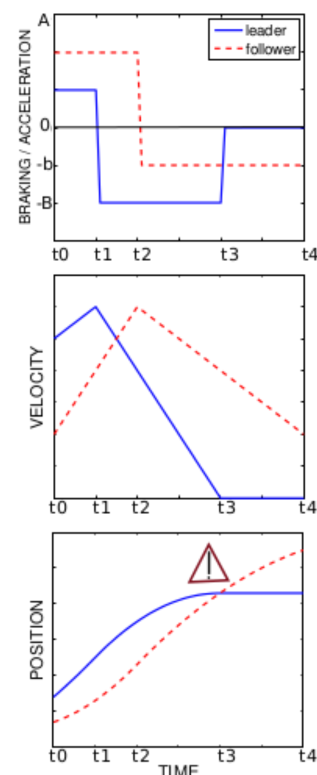

Abbildung 5.4: Darstellung der ACC-Abläufe in der Fallstudie

Im initialen Zustand befindet sich das erste Fahrzeug 40 Meter vor dem zweiten Fahrzeug. Beide Fahrzeuge besitzen eine Geschwindigkeit von $120 km/h$ ($33 m/s$). Anfangs beschleunigen beide Fahrzeuge mit $1 m/s^2$.

Damit ist das System von Differentialgleichungen gegeben durch:

$$\dot{x} = \begin{pmatrix} \dot{p}_1 = v_1 \\ \dot{v}_1 = a_1 \\ \dot{a}_1 = 0 \\ \dot{p}_2 = a_1 \\ \dot{v}_2 = a_2 \\ \dot{a}_2 = 0 \end{pmatrix}$$

Sowie durch die folgenden Anfangsbedingungen:

$$x_0 = \begin{pmatrix} p_1(0) \\ v_1(0) \\ a_1(0) \\ p_2(0) \\ v_2(0) \\ a_2(0) \end{pmatrix} = \begin{pmatrix} 40 \\ 33 \\ 1 \\ 0 \\ 33 \\ 1 \end{pmatrix}$$

Abschließend soll das Szenario als hybrider Automat modelliert werden. Eine mögliche Umsetzung ist in Abbildung Abbildung 5.5 dargestellt. Das Szenario wird durch vier Zustände modelliert.

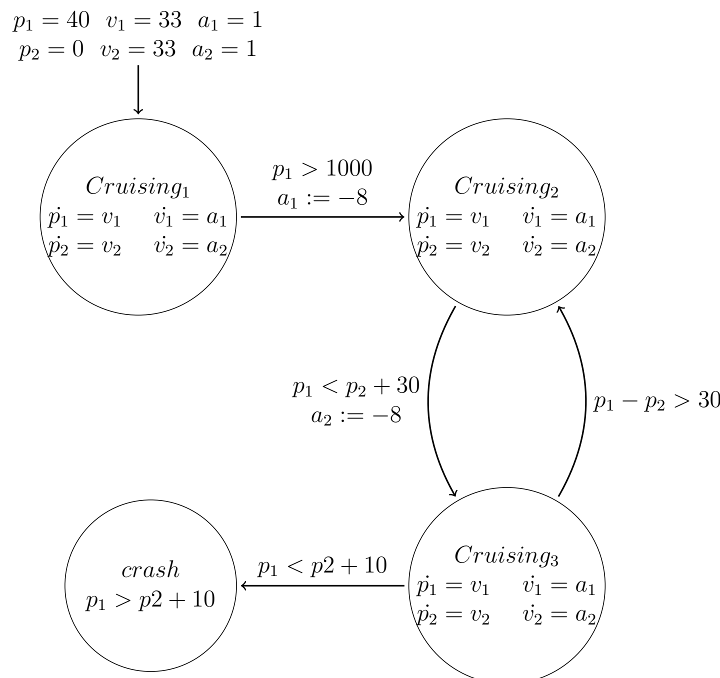

Abbildung 5.5: Hybrider Automat der Adaptive Cruise Control

Drei Zustände modellieren das Fahren. Fahrzustand 1 ($Cruising_1$) modelliert das Fahren der beiden Fahrzeuge bis zur 1000 Meter Marke. In Fahrzustand 2

($Cruising_2$) bremst das vorausfahrende Fahrzeug, sofern die 1000 Meter überschritten wurden. Für das hinterherfahrende Fahrzeug wird die vorherige Beschleunigung übernommen. In Fahrzustand 3 ($Cruising_3$) hat sich das hinterherfahrende Fahrzeug soweit genähert (nähere als 20 Meter zum vorausfahrenden Fahrzeug), dass es ebenfalls zu bremsen beginnt.

Die Verzögerung (negative Beschleunigung) wurde für dieses Beispiel mit $8m/s^2$ festgelegt. Dies entspricht einer Vollbremsung auf flacher und asphaltierter Strecke.

Der Zusammenstoß beider Fahrzeuge erfolgt, wenn die Position des Fahrzeugs 2 größer ist als die von Fahrzeug 1. Außerdem wird die Fahrzeuglänge beachtet, da die Position die Position der Spitze darstellt. Hier wurden beispielhaft zehn Meter Fahrzeuglänge gewählt.

Eine Umsetzung des Modells in SAML ist in Abschnitt A.5 zu finden. Das Modell betrachtet das Szenario für einen Zeitraum von einer Minute. In dieser Modellierung wurde darauf verzichtet drei Fahrzustände einzuführen. Stattdessen gibt es zwei Zustände (Fahren und Crash). Die Änderungen der Beschleunigungsparameter werden über Zustandsübergange innerhalb des Fahrzustands realisiert.

5.2.2 Veranschaulichung der Änderung von Parametern im Modell

Das eingeführte Beispiel der Adaptive Cruise Control dient im Folgenden dazu dazu die in eingeführte Behandlung von sich wechselnden Parametern zu erläutern.

Auf Abbildung 5.6 ist das Ausgangsverhalten des Systems für eine Betrachtung von 60 Sekunden in einem Funktionsgraph von Zeit zu Position der beiden Fahrzeuge aufgetragen. Die Berechnungen wurden hier für die Graphen mit einem zeitlichen Abstand von $\Delta t = 0.1$ durchgeführt.

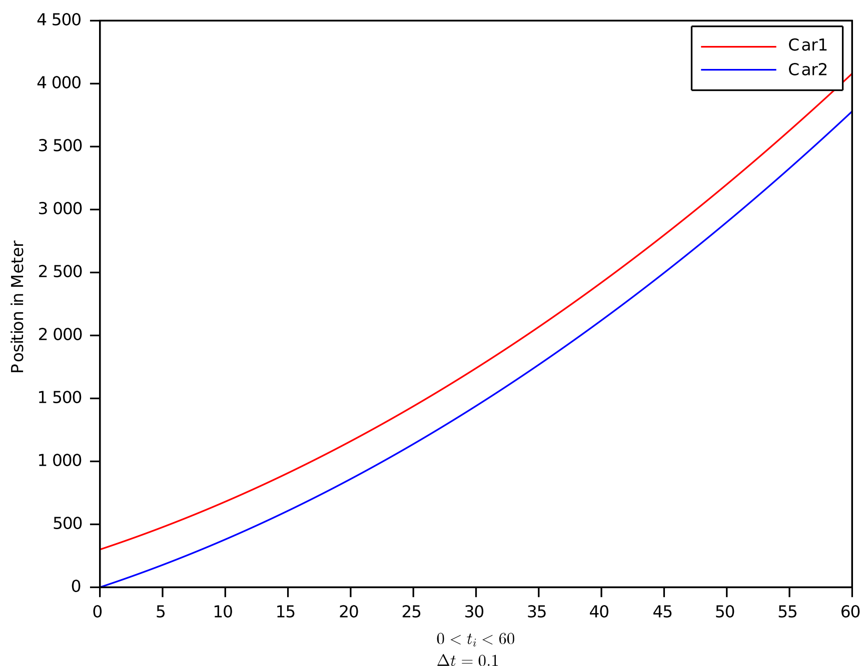

Abbildung 5.6: Berechnete Position der beiden Fahrzeuge für Ausgangsbedingungen

Über die Zeit erhöht sich die Position beider Fahrzeuge. Da beide Fahrzeuge im Initialzustand dieselbe Geschwindigkeit und Beschleunigung besitzen, beträgt der Abstand durchgehend 300 Meters und es würde, sofern sich die Parameter der Beschleunigung ändern würden, nicht zu einem Zusammenstoß kommen.

Nach 1000 Metern beginnt das vorausfahrende Fahrzeug mit $8m/s^2$ zu bremsen. Dieses Verhalten wird im SAML-Modell durch ein Gegenbeispiel festgestellt, da nach 17.1 Sekunden das erste Fahrzeug mehr als 1000 Meter gefahren ist, wird im Modell der Parameter der Beschleunigung reduziert. Dies wird als folgendes Gegenbeispiel von dem Modelchecker zurückgegeben:

$$a_1 = -8 \qquad a_2 = 1 \qquad state = cruise \qquad CLOCK.TIME = 171$$

Da der neue Zustand nicht der Crashzustand ist, wird ein Wechsel der Parameter durchgeführt und eine Neuberechnung des Models vom Zeitpunkt $t_n = 171$.

Auch die nun betrachtete Spezifikation ändert sich, nachdem die Parameter der Beschleunigung jeweils angepasst wurden. Es ergibt sich folgende Spezifikation:

$$!(\mathrm{EF}(state = CRASH \vee a_2 \neq -8 \vee a_2 \neq 1));$$

Das durch das neu transformierte Modell entstandene Verhalten ist graphisch in Abbildung 5.7 wieder als Funktionsgraph dargestellt.

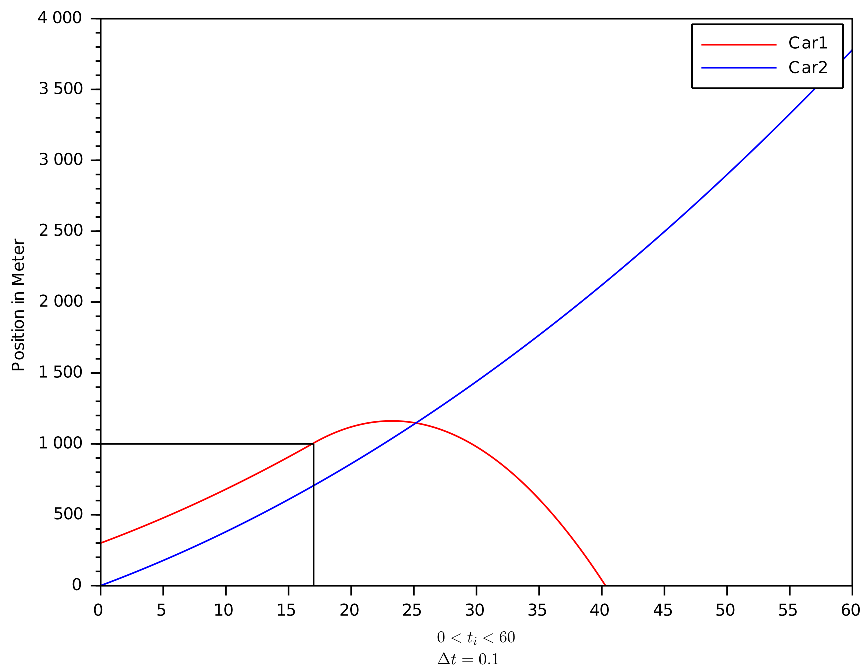

Abbildung 5.7: Berechnete Position der beiden Fahrzeuge bei erstem Bremsen

In dem Graphen ist das Einsetzen des Bremsvorgangs eingezeichnet. Aus dem Graphen ist außerdem abzulesen, dass das Fahrzeug nach ca. 25 Sekunden beginnen würde zurückzufahren, was ein Resultat der negativen Beschleunigung ist.

Da dies nicht modelliert werden soll, sondern das Fahrzeug in dem Fall stehen bleiben bzw. langsam weiterfahren soll, ist im Modell festgelegt, dass bei einer Geschwindigkeit, die geringer als $2m/s$ ist, die Beschleunigung auf Null gesetzt wird. Diese Korrektur setzt im Modell nach 23 Sekunden ein.

Ab diesem Zeitpunkt steht das Fahrzeug bzw. bewegt sich mit geringer Geschwindigkeit fort, während das hinterherfahrende Fahrzeug sich diesem nähert. Nach 25.1 Sekunden ist das hinterherfahrende Fahrzeug bis auf 20 Meter an das vorausfahrende Fahrzeug herangefahren, sodass auch dieses Fahrzeug anfängt zu bremsen.

In Abbildung 5.8 sind sowohl der Zusammenhang der Beschleunigungsanpasssung auf $0m/s^2$ für das vorausfahrende Fahrzeug als auch die Anpassung der Verzögerung für das hinterherfahrende Fahrzeug dargestellt.

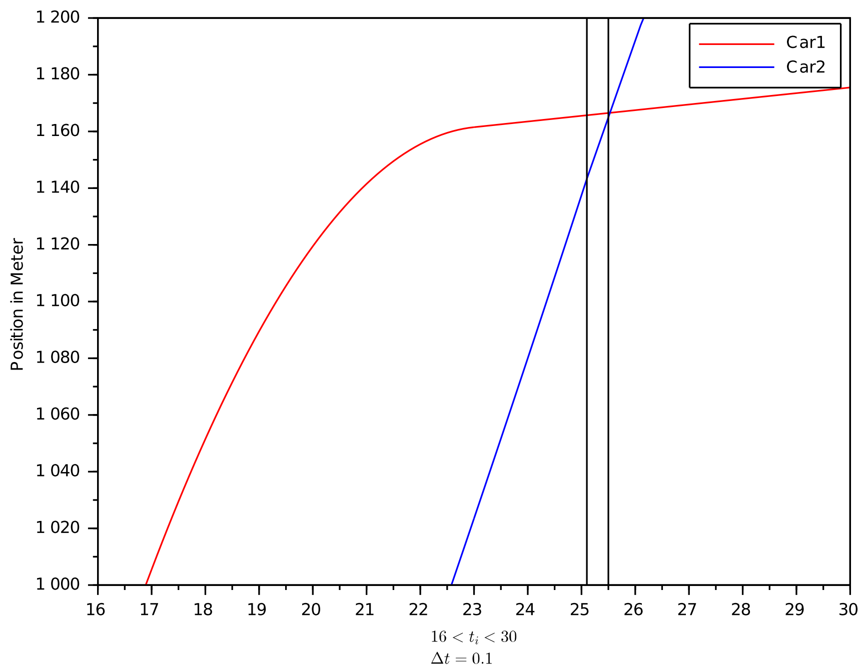

Abbildung 5.8: Crashverhalten der beiden Fahrzeuge nach Bremsen des vorausfahrende Fahrzeugs

Auf dem Funktionsgraphen ist zu erkennen, dass nach dem Einsetzen der Bremsung bei Fahrzeug 2 zu wenig Zeit vorhanden ist, um einen Crash zu vermeiden. Zu diesem kommt es dann nach 25.8 Sekunden. Für diesen Fall gibt der Modelchecker folgendes Gegenbeispiel für das Erreichen des Crash-Zustands aus:

$$a_1 = 0 \qquad a_2 = -8 \qquad state = crash \qquad CLOCK.TIME = 258$$

Aus diesem Gegenbeispiel können nun die Informationen herausgelesen werden, wie die Parameter für die Beschleunigung belegt sind. Außerdem können die Geschwindigkeit und die Position aus der gelösten Differentialgleichung abgeleitet werden.

Der Verlauf von Geschwindigkeit und Position kann nun über die jeweiligen Wertparameter der Differentialgleichungen von Position und Geschwindigkeiten bis zu den Zeitpunkten der Parameterwechsel gezeichnet werden.

In Abbildung 5.9 ist der gesamte Verlauf dargestellt für die beiden Fahrzeuge dargestellt.

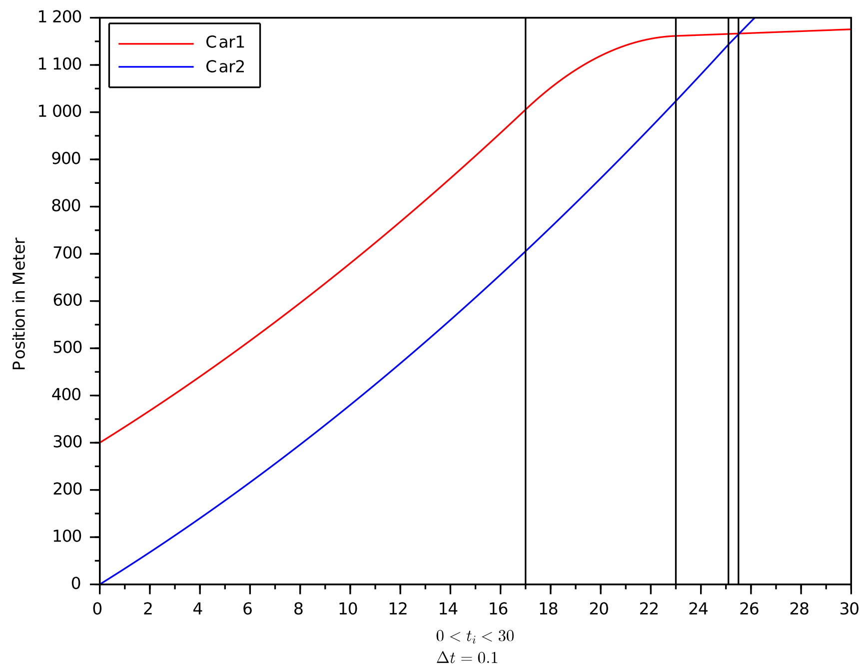

Abbildung 5.9: Gesamter Ablauf der ACC-Fallstudie

Durch die senkrechten Linien sind die drei Parameterwechsel ($t = 17,1$, $t = 23$ und $t = 25,1$) sowie die Änderung des Zustands, der den Crash signalisiert, gekennzeichnet. Also die vier Stellen, in denen der Model Checker ein Gegenbeispiel ausgibt.

6. Feststellung der Eigenschaften des entwickelten Verfahrens anhand von Experimenten

Anhand von mehreren Experimenten sollen in diesem Kapitel Eigenschaften des Transformer festgestellt werden. Außerdem soll der Transformer mit dem hybriden Model Checker iSAT verglichen werden. Diese Ergebnisse dienen folgenden Kapitel im Anschluss dazu, die Arbeit bewerten zu können. Das Kapitel führt zuerst ein, unter welchen Bedingungen die Ergebnisse gewonnen wurden. Anschließend werden Experimente anhand der beiden vorgestellten Fallstudien durchgeführt.

6.1 Testumgebung

Damit die gewonnen Ergebnisse der Experimente nachvollziehbar sind, wird in diesem Abschnitt die Testumgebung beschrieben.

Die Experimente wurden alle auf Ubuntu 15.04 mit fogender Rechnerkonstellation durchgeführt:

- Prozessor: Intel® Cor i7-5500U (2,4 GHz)
- Arbeitsspeicher: 8 GB DDR3 1600MHz

VECS wurde mit dem zu dieser Arbeit zugeordneten Branch mit dem Stand vom 28.04.2015 durchgeführt. Scilab wurde unter der Version 5.5.1 mit den zugehörigen Java Bibliotheken für Linux verwendet. Die Experimente mit iSAT wurden mit iSAT in der Version 1.0r1322 durchgeführt.

iSAT wendet einen Bounded Model Checking Ansatz an (siehe Abschnitt 2.5). In dem in dieser Arbeit vorgestellten Prototypen, wird CTL Model Checking eingesetzt. Daher ist beim Vergleich dieser beiden Techniken darauf zu achten. Ein Vergleich ist trotzdem möglich, weil es sich auch bei dem hier geschilderten Erreichbarkeitsproblem, eine Art von Grenze durch die Zeit t_{end} exisitiert.

6.2 Punktförmige Zugbeeinflussung

Das Modell der punktförmigen Zugbeeinflussung dient im ersten Teil zum Vergleich mit dem in [Nie13] vorgestellten Ansatz. Dazu wird das in der Arbeit vorgestellte Experiment mit dem in dieser Arbeit eingeführten Verfahren durchgeführt.

Im zweiten Teil werden für die gegebene Implementierung die einzelne Teile der Transformation hinsichtlich der Laufzeit analysiert.

6.2.1 Vergleich mit anderen Ansätzen

Das Modell der punktförmigen Zugbeeinflussung soll im ersten Experiment mit dem in [Nie13] vorgestellten Ansatz verglichen werden.

Dort wurde die Fallstudie der Zugbeeinflussung verwendet, um das dort vorgestellte Verfahren hinsichtlich der Performance zu testen. Dafür wurden die Aspekte des aufgebauten Zustandsraum und der Laufzeit betrachtet. Der aufgebaute Zustandsraum beeinflusst direkt die Laufzeit des Model Checks.

Für die Fallstudie wurden zwei Spezifikationen betrachtet:

- A: $\neg \boldsymbol{EF}(v \geq 25 \wedge p > 100)$
- B: $\neg \boldsymbol{EF}(v \geq 25 \wedge p > 120)$

Also ob der Zug jemals eine höhere Geschwindigkeit als $25[m/s]$ besitzt, sobald er eine festgelegte Position überfahren hat. In dem Beispiel wurden als 100 (Fall A) und 120 Meter (Fall B) als Grenzen festgelegt. In Tabelle 6.1 ist für fünf Ansätze das jeweilige Ergebnis für die Spezifikation A oder B sowie die Laufzeit angegeben.

Werkzeug	Fall	Laufzeiten	Ergebnis
OpenModelica	A	$<$ 5 Sekunden	unsicher
	B	$<$5 Sekunden	sicher
HSolver	A	$>$ 5 Stunden	unklar
	B	$>$5 Stunden	unklar
[Nie13]	A	$>$ 12 Stunden	keine Angabe, Abbruch
	B	$>$12 Stunden	keine Angabe, Abbruch
SAML/NuSMV	A	5 Sekunden	unsicher, keine feste Aussage möglich
	B	5 Sekunden	„sicher“, keine Aussage möglich
iSAT	A	$<$ 1 Sekunde	unsicher
	B	$<$ 1 Sekunde	„sicher“, keine Aussage möglich

Tabelle 6.1: Vergleich der Laufzeiten und Ergebnisse beim Prüfen der Fälle A und B mit $\Delta t = 0,1$ (die ersten drei Zeilen sind [Nie13] entnommen)

Die ersten drei betrachteten Ansätze wurden [Nie13] entnommen. Bei OpenModelica handelt es sich um keinen Model Checker, sondern um eine numerische Lösung. Hier wurde die Differentialgleichung gelöst und anschließend graphisch aufgetragen. Die beiden Bedingungen für Geschwindigkeit und Position können nun jeweils graphisch überprüft werden.

Für den Fall A ist das Modell unsicher, da der Zug nach 100 Metern nicht genug Wegstrecke besaß, um auf eine Geschwindigkeit von unter $25[m/s]$ abzubremsen. Für eine Strecke von 120 Metern ist das Modell sicher.

HSolver ist der in dieser Arbeit als Vergleich betrachtete Modelchecker. Dieser konnte genauen Aussagen zu dem Modell treffen. Wie auch der in [Nie13] vorgestellte Ansatz. Das Problem lag daran, dass der Zustandsraum für das Modell sehr groß wurde, um die Lösung der Differentialgleichung zu modellieren. Deshalb ist auch automatisch die Dauer angestiegen, in der das Modell gelöst wurde.

Der in dieser Arbeit umgesetzte Prototyp konnte das Modell für den Fall A in 5 Sekunden als unsicher einordnen. Trotzdem ist diese Aussage nicht bindend, weil Scilab keine genaue Fehlerberechnung durchgeführt hat, sondern nur eine Toleranzeinschtzung (siehe Abschnitt 4.3.1). Für Fall B konnte der Ansatz das Modell als sicher einstufen. Hier ist jedoch zu beachten, dass diese Aussage nicht zwingend wahr sein müsste, da das Verhalten zwischen den Diskretisierungsschritten nicht näher betrachtet wurde.

Diese Arbeit hat das Modell ebenso mit iSAT geprüft. Das zugehörige Modell ist in Abschnitt A.3 im Anhang zu finden. Die Ergebnisse sind identisch zu denen in dieser Arbeit gewonnenen. Die Laufzeit ist jedoch geringer.

6.2.2 Experiment zur Performance der Transformationsprozesse

Im vorherigen Abschnitt wurde bereits gezeigt, dass der vorgestellte Ansatz mit etwa 5 Sekunden für das betrachtete Modell ein wenig langsamer ist als ein hybrider Bounded Modelchecker wie iSAT. Deswegen sollen nun alle Prozesse der in Abschnitt 4.4 beschriebene Abläufe in ihrer Laufzeit in Relation zur Größe des Modells untersucht.

Die Größe des Modells ist durch die Diskretisierungsstufen Δt gegeben. Für große Δt ist das Modell klein, da nur wenige Diskretisierungen durchgeführt werden. Für kleine Δt entstehen viele Diskretisierungspunkte und somit ein Modell mit vielen Verknüpfungen innerhalb der Übergangsbedingungen.

Die Laufzeit wird innerhalb der Implementierung durch Festhalten der Systemzeit in Millisekunden vor und nach Ausführung des jeweiligen Ablaufs aufgenommen. Die Laufzeit ergibt sich als Differenz beider Zeiten.

Tabelle 6.2 beschreibt für verschiedene Diskretisierungszeiten Δt die Laufzeit der einzelnen Abläufe.

Δt	Initialisierung	Lösen DG	Transformation	NuSMV	Sonstige	$\sum$
1	1,35	0.05	0,65	1,08	0,1	3,22
0,5	1,36	0,07	0,94	1,08	0,06	3,53
0,25	1,40	0.06	1,50	1,09	0,13	3,53
0,1	1,35	0.05	2,55	1,12	0,34	5,43
0,05	1,40	0.06	4,18	1,11	0,55	7,23
0,025	1,40	0,07	6,596	1,17	0,55	9,79
0,01	1,38	0.05	11,87	1,41	1,03	15,68

Tabelle 6.2: Laufzeiten (in Sekunden) des Indusi-Modells für verschiedene Diskretisierungsstufen Δt

Die Initialisierung des Modells ist der erste Ablauf. Hierzu gehört, dass das Modell zuerst aus einer Textdatei gelesen und in eine Modellklasse importiert wird.

Anschließend werden aus dem Modell das Zeitsystem und die Differentialgleichungen herausgelesen. Diese Phase besitzt für alle Δt etwa dieselbe Laufzeit von ca. 1,5 Sekunden. Eine nähere Betrachtung dieser Vorgänge hat ergeben, dass vor allem das Lesen aus dem Dateisystem und transformieren des Modells in das zugehörige Klassensystem für SAML-Modelle nahezu die volle Laufzeit in Anspruch nimmt.

Das Lösen der Differentialgleichungen über das Scilab-Interface nimmt für alle Δt kaum Zeit in Anspruch. Hier hat sich erst testweise für $\Delta t << 0,001$ herausgestellt, dass Scilab eine größere Menge an Arbeitsspeicher benötigt, um eine Lösung zu finden.

Das Modelchecking in NuSMV nimmt in etwa eine Sekunde in Anspruch. Hier wurde auch die Zeit eingerechnet, die benötigt wird, um das Modell innerhalb von VECS nach NuSMV zu übersetzen.

Unter dem Punkt Sonstige wurden weitere Abläufe im Modell zusammengefasst wie die Prüfung des Modells auf Indeterminismen.

Den Hauptanteil der Laufzeit ergibt sich für die kleineren Δt durch die Transformation des Modells. Unter diesem Ablauf wurde die Detektion von fehlerhaften Übergängen und die Anpassung alle Übergänge durch Einfügen der Referenz auf den zeitlichen Automat zusammengefasst.

Die Entwicklung dieser Laufzeit für kleiner werdende Δt ist zur besser Veranschaulichung in Abbildung 6.1 dargestellt.

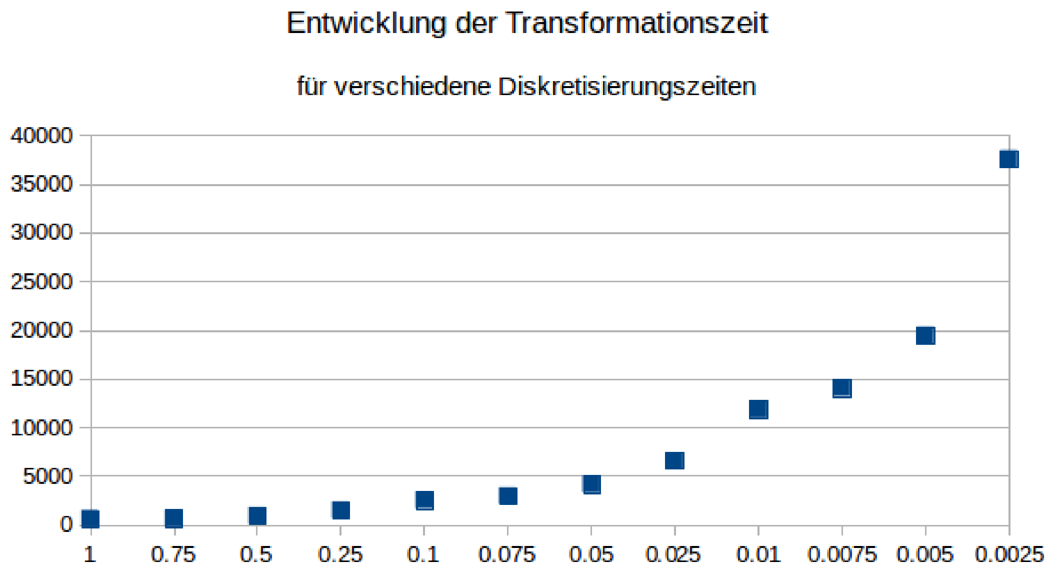

Abbildung 6.1: Entwicklung der Transformationszeit für verschiedene Diskretisierungszeiten Δt

Die jeweiligen Zeiten Δt sind auf der x-Achse in einer logarithmischen Skala aufgetragen. Auf der y-Achse sind die Laufzeiten in Millisekunden aufgetragen.

Aus dieser Abbildung ist zu erkennen, dass für kleiner werdende Δt die Laufzeit dramatisch ansteigt. Für $\Delta t = 0,0025$ ergibt sich bereits eine Laufzeit von fast 40 Sekunden. Dies entspricht bei dem Betrachtungszeitraum $t_{end} = 5[sec]$ einer Anzahl von $t_n = \frac{t_{end}}{\Delta t} = \frac{5}{0{,}0025} = 2000$ Diskretisierungsschritten.

Eine Ursache für diese Entwicklung liegt in der Implementierung der Transformation. Für die Injektion der jeweiligen Lösungswerte der Differentialgleichungen in die Übergangsbedingungen, werden immer wieder neue Modellobjekte erstellt. Dies ist wie bereits bei der Initialisierung des Systems sehr ineffektiv, da auf das Dateisytem zugegriffen wird. Bei einer großen Menge an Diskretisierungsschritten erhöht sich die Anzahl der Erzeugung von Modellobjekten und somit auch die Laufzeit der Transformation.

6.3 Adaptive Cruise Control

Wie bereits in Abschnitt 5.2 soll die Adaptive Cruise Control mit zwei Experimenten dazu dienen, das Verhalten eines parametrisierten Modells zu untersuchen. Im ersten Experiment untersucht ein Experiment die Laufzeiten des Modells nach jeder Iteration, in der ein Parameter im Modell verändert wurde. Ein zweites Experiment vergleicht im Anschluss den in dieser Arbeit vorgestellten Ansatz für das Modell der Adaptive Cruise Control mit iSAT.

6.3.1 Verhalten der Adaptive Cruise Control bei wechselnden Parametern

Das in dieser Arbeit betrachtete Modell der Adaptive Cruise Control verändert, wie in Abschnitt 5.2 gezeigt wurde, mehrfach die Parameter. Die Änderungen der Parameter sollen im Folgenden als Laufzeiten durch P_1, P_2, P_3, P_4 gekennzeichnet werden.

Dabei haben diese Parameterwechsel folgende Bedeutung:

- P_1: Laufzeit aus der Ausgangskonstellation (siehe: Abschnitt 5.2.1) bis zum ersten Parameterwechsel
- P_2: Laufzeit von der Änderung des Parameters $a_1 = 1$ auf $a_1 = -8$ (erste Bremsung, siehe: Abbildung 5.7) bis zum nächsten Parameterwechsel
- P_3: Laufzeit von der Änderung des Parameters $a_1 = -8$ auf $a_1 = 0$ (Anpassen der Geschwindigkeit) bis zum nächsten Parameterwechsel
- P_4: Laufzeit von der Änderung des Parameters Änderung des Parameters $a_2 = 1$ auf $a_2 = -8$ (Bremsen des hinterherfahrenden Fahrzeugs, siehe: Abbildung 5.8) bis zum nächsten Parameterwechsel

In dem nun folgenden Experiment soll die Laufzeit der Transformationen für jedes entstehende Modell untersucht werden. Außerdem soll gleichzeitig der Einfluss der vorhandenen Diskretisierungsschritte betrachtet werden. Nach Abschnitt 3.2.1 ergibt sich die Anzahl der betrachteten Diskretisierungsschritte t_n aus $t_n = \frac{t_{end}}{\Delta t}$.

Tabelle 6.3 stellt jeweils für P_1, P_2, P_3, P_4 bei zwei betrachtete Grenzen $t_{end1} = 30$ und $t_{end2} = 60$ (die Einheit entspricht jeweils Sekunden im Modell) und verschiedenen Δt die Laufzeit des Prototypen dar. Das Ziel besteht darin, die Anzahl der entstehenden Diskretisierungsschritte für t_{end1} und t_{end2} gleich ist. Deshalb ist für t_{end2} das jeweils gewählte Δt doppelt so groß als t_{end1}

Die Tabelle zeigt, wie zu erwarten war, dass $P_4 < P_3 < P_2 < P_1$ (unabhängig von dem gewählten t_n). Dieses Verhalten ist insofern erwartbar, da sich nach jedem

t_n	t_{end}	Δt	P_1	P_2	P_3	P_4	$\sum$
30	30	1	8,6	5,3	2,4	2	18,3
60	30	0,5	11,4	5,6	4,3	4,2	25,5
	60	1	12,0	5,8	5,4	4,4	27,6
120	30	0,25	17,1	9,2	8,4	6,3	41,0
	60	0,5	17,7	9,8	8,5	8,4	44,4
240	30	0,125	29,6	16,1	14,4	12	72,1
	60	0,25	30,5	16,8	16,8	16,4	80,3

Tabelle 6.3: Laufzeiten (in Sekunden) des ACC-Modells für verschiedene Diskretisierungsstufen Δt und maximalen Zeitpunkten t_n

Parameterwechsel zu einem Zeitpunkt t_i die Menge der Diskretisierungsschritte um die bereits zum Zeitpunkt t_i durchgeführte Diskretisierung, reduziert.

Ein interessanter Aspekt, der sich aus dieser Tabelle gewinnen lässt, ist die geringere Laufzeit für den kleineren Betrachtungszeitraum t_{end1} im Gegensatz zum größeren Betrachtungszeitrum t_{end2}. Dies lässt sich für das Modell dadurch erklären, dass alle Parameterwechsel im Bereich von 15 bis 25 Sekunden im Modell stattfinden (siehe Abbildung 5.8).

Für den längeren Betrachtungszeitrum bis 60 Sekunden ergibt sich somit nach allen Parameterwechseln ein größerer Diskretisierungszeitraum von mindestens über 30 unnötigen Sekunden, während bei dem kleineren Betrachtungszeitraum dieser Zeitraum bei 5-15 Sekunden liegt. Da der Transformer umso länger benötigt, wenn mehr Disktreisierungsschritte vorhanden sind, ist die Laufzeit für den größeren Betrachtungszeitraum und für mehrere Parameterwechsel erkennbar größer.

6.3.2 Vergleich mit iSAT

Abschließend soll der vorgestellte Ansatz für die Adaptive Cruise Control mit iSAT verglichen werden.

Dazu wird das Modell der ACC dahingehend verändert, dass das Einsetzen des Bremsvorgangs des vorausfahrenden Fahrzeugs variabel gestaltet wird, um das Verhalten beider Ansätze in bezüglich der zeitlichen Performance vergleichen zu können.

Da für den in dieser Arbeit vorgestellten Ansatz immer ein Zeitraum angegeben werden muss, für den das Modell berechnet wird, betrachtet das Experiment zwei Fälle für den Ansatz:

1. „Best Case“: Für diesen Fall wird als t_{end} der kleinste Zeitpunkt gewählt, in dem das Modell mit einem Gegenbeispiel berechnet wird.

2. „Bad Case“: In diesem Fall wird t_{end} mit 60 Sekunden festgelegt. Dadurch sind im Modell viele unnötige Berechnungsschritte enthalten, die nicht benötigt werden, um zu dem Gegenbeispiel zu gelangen.

In Tabelle 6.4 ist der Vergleich bei verschiedenem Einsetzen des Bremsverhaltens mit iSAT und dem jeweiligen Fall für die hier gegebene Umsetzung dargestellt.

An dieser Stelle sei nochmal angemerkt, dass iSAT ein Bounded Model Checking Verfahren anwendet. Dies resultiert in dem Ergebnis darin, dass für Modelle mit einem größeren t_n die Berechnungsdauer anwächst, da jeweils die betrachtete Grenze

Einsetzen Bremsen [m]	t_n	iSAT	„Best Case“ SAML/NuSMV	„Bad Case“ SAML/NuSMV
300	101	1,9	40,0	Out Of Memory
500	149	5,3	40,2	Out Of Memory
1000	255	16,5	49,7	Out Of Memory
2000	423	44,8	65,8	378,5
3000	558	85,7	83,4	173,2
5000	777	169,8	114,5	140,0
6000	870	192,0	124,5	124,5

Tabelle 6.4: Laufzeiten (in Sekunden) des ACC-Modells für iSAT und SAML/-NuSMV für verschiedene Bremsstufen ($\Delta t = 0.1$)

für jeden Schritt größer wird und somit pro Schritt ein größeres Modell betrachtet wird.

Für den „Best Case“ liegt die hier betrachtete Lösung deutlich unter der Performance von iSAT. Dies liegt daran, dass das Modell viermal neu berechnet wird und jeweils viel Zeit intern für das Erstellen der neuen Vergleiche benötigt wird.

Setzt das Bremsen erst später ein (z.B. nach 5000 Metern oder 6000 Metern) bietet der „Best Case“ jedoch eine bessere Performance von iSAT. Das begründet sich dadurch, dass hier die Parameterwechsel sehr spät im Modell erfolgen. Deshalb muss nach jedem Parameterwechsel nicht mehr fast das gesamte Modell neu berechnet werden, sondern nur kleinere Teilschritte mit wenigen Diskretisierungsschritten.

Im hier betrachteten „Bad Case“ wirkt sich sich Erhöhung des gewählten $t_n = 900$ dramatisch auf die betrachteten Fälle aus, dass das Bremsen früh einsetzt. In diesen Fällen bricht die Implementierung im Test sogar ab, da zu wenig Speicher vorhanden ist. Dies liegt daran, dass in der Implementierung für alle Modelle Speicher verwendet wird und der benötigte Speicher ab vier sehr großen Modellen wie sie für frühe Einsetzen des Bremsvorgangs entstehen sehr groß ist.

Für das Bremsen nach 2000 Metern zeigt sich wie stark sich die Performance in diesem Fall verschlechtert. Benötigt der Transformer hier im „Best Case“ etwas mehr als eine Minute, so sind es in dem schlechten Fall über 6 Minuten.

Dieses Experiment hat gezeigt, dass sich der hier vorgestellte Transformer für Modelle eignet, in denen sich die Parameter erst ändern, wenn bereits ein großer Teil des Modells abgearbeitet wurde. Im Vergleich zu iSAT ist er eher ungeeignet für Modelle, die Parameterwechsel vor allem zu Beginn der Berechnungen durchführen. Hier zeigt sich auch, dass die Grenze t_{end} klug gewählt werden sollte, um die Performance nicht zu stark zu beeinträchtigen.

7. Bewertung des Verfahrens

Es sollte in dieser Arbeit ein Verfahren entwickelt werden, das gewöhnliche Differentialgleichungen 1.Ordnung über einen externen Modelchecker löst, diese Lösung in ein Modell einfügt, sodass das Modell mit einem symbolischen Modelchecker gelöst werden sollte. Das Ziel bestand darin herauszufinden, welche Stärken und Schwächen dieser Ansatz bringt.

Ein solches Verfahren wurde in Kapitel 3 vorgestellt und eine Umsetzung, die in Kapitel 4 beschrieben wurde, realisiert.

Nun soll anhand des vorgestellten Verfahrens und dessen Umsetzung und den Ergebnissen aus dem vorherigen Kapitel in diesem Kapitel das vorgestellte Verfahren hinsichtlich Stärken und Schwächen bewertet werden. Anschließend geht dieses Kapitel darauf ein, welche Verbesserungsmöglichkeiten es für das Verfahren gibt und welche offenen Fragen es für anschließende Arbeiten für das Verfahren existieren.

7.1 Beurteilung der Eigenschaften

In diesem Abschnitt soll eine Analyse der Eigenschaften des in dieser Arbeit eingeführten Verfahrens durchgeführt werden.

Zuerst werden die Stärken des vorgestellten Verfahrens analysiert. Im Anschluss die Schwächen.

Im Folgenden werden mehrere Stärken des Transformationsverfahrens und im Zusammenhang auch des umgesetzten Prototyps diskutiert:

- **Schnelligkeit der Nutzung einer Schnittstelle wie Scilab:** In dem Prototypen hat sich die Schnittstelle zu Scilab als sehr effizient herausgestellt (siehe Zeiten für das Lösen von Differentialgleichungen in Tabelle 6.2). Dadurch

- **Performance:** Im Vergleich zu dem in [Nie13] vorgestellten Ansatz, profitiert diese Lösung davon, dass der Zustandsraum klein gehalten wird und nicht wie in [Nie13] dadurch vergrößert wird, dass die Lösung der Differentialgleichung in das Modell abstrahiert wird.

  In den beiden Fallstudien hat sich gezeigt, dass sich die Ergebnisse, ob eine Spezifikation unsicher ist oder ggf. sicher ist, innerhalb von akzeptablen Zeiträumen finden lässt. Das bedeutet, dass für die zwei Fallstudien alle Laufzeiten im Minutenbereich lagen.

- **Parameterwechsel** In Tabelle 6.4 wurde gezeigt, dass für einen Parameterwechsel in der Nähe von größeren t_{end} im Vergleich zu iSAT die Laufzeit ähnlich oder sogar besser ist.

Demgegenüber haben sich folgende Schwächen herausgestellt:

- **Keine Fehlerberechnung in Scilab:** Wie bereits in Abschnitt 4.3.1 erläutert wurde, nutzt Scilab keine explizite Fehlerberechnung, sondern schätzt einen Fehlerwert, der nicht überschritten werden darf. Dadurch kann im Endeffekt für ein Gegenbeispiel, das der Model Checker liefert nicht für alle Fälle ausgeschlossen werden, dass dieses Gegenbeispiel immer gültig ist.

- **Umformung der Zustandsbedingungen in VECS:** Das Experiment, das Tabelle 6.2 beschreibt, hat gezeigt, dass ein großer Teil der Laufzeit des Model Checking Prozess' daraus besteht, das Modell zu transformieren. Wie in Abschnitt 6.2.2 beschrieben wurde, liegt die Ursache im Erstellen der Vergleiche für jeden Diskretisierungsschritt.

- **Indeterminismen:** Das Laufzeitverhalten für das Aufbauen der Vergleiche wirkt sich negativ auf den gesamten Prozess aus. Bei Indeterminismen im Modell muss der Transformationsprozess mehrfach ausgeführt werden.

  Ein Schwachpunkt des Prototyps besteht momentan darin, dass er ausschließlich alle Modelle zurückgibt, die bei einem Modell mit Indeterminismen betrachtet werden müssen.

- **Parameterwechsel:** Sofern ein Modell viele Parameterwechsel enthält, wird sich das negativ auf die Performance auswirken. Es hat sich vor allem gezeigt, dass für ein Modell, dass Parameterwechsel nach wenigen Diskretisierungsschritten, die Laufzeit stark anwächst, sofern t_n sehr groß ist (siehe: Tabelle 6.4).

  Die Lösung des Parameterwechsel hat den Nachteil, dass explizit ein Prozess für einen Model Checker implementiert werden muss, um das Ergebnis des Model Checkers auszuwerten.

Trotz dieser Schwächen lässt sich feststellen, dass aus dem in dieser Arbeit vorgestellten Verfahren und dem umgesetzten Prototypen einige Erkenntnisse gewonnen werden konnten. Im Folgenden sollen Verbesserungsmöglichkeiten für den vorgestellten Prototypen vorgestellt werden.

7.2 Verbesserung der Implementierung

Der folgende Abschnitt geht auf drei Punkte konkret ein, wie der umgesetzte Prototyp verbessert werden kann. Diese drei Punkte sind nur eine Auswahl an möglichen Ansätzen

7.2.1 Behandlung von Indeterminismen

Ein wichtiger Punkt ist, dass der Aspekt des Backtrackings bei Indeterminismen in dem Model Checking Prozess umgesetzt wird. Momentan werden, wie bereits

beschrieben wurde, ausschließlich für den Fall, dass Indeterminismen im Modell vorkommen, mehrere Modelle zurückgeben, die alle möglichen Folgebelegungen abdecken. Für diese muss jeweils ein separater Model Checking Prozess gestartet werden. Dieser Ablauf könnte in den bisherigen Ablauf des Model Checking Prozess' integriert werden.

Eine weitere Überlegung könnte es sein, dass für jeden durch einen Indeterminismus entstehenden Zweig der aktuelle Systemzustand gespeichert wird, um so den Backtracking-Algorithmus anwenden zu können, wenn ein Zweig die Ausgangsspezifikation bestätigt. Dafür ist ein Zugriff auf den Systemzustand durch den symbolischen Modelchecker nötig. Hier müsste herausgefunden werden, wie sich dieser auslesen lassen könnte.

7.2.2 Erstellen von Vergleichen in VECS

Die Experimente im vorherigen Kapitel haben gezeigt, dass beim Transformationsprozess die Performance bei vielen Diskretisierungsschritten immer schlechter wird. Eine Ursache dafür ist, dass für jeden einzelnen Vergleich zuerst ein neues SAML-Modell angelegt wird, wofür wiederum in das Dateisystem geschrieben wird. Ein wichtiger Schritt in der Implementierung ist es einen Weg zu finden, mit dem sehr effektiv ein Element wie ein Vergleich aus einer Zeichenkette erstellt werden kann. Die Möglichkeiten sollten innerhalb von VECS dafür vorhanden sein, da es sich um ein reines Erstellen von Objekten handelt.

7.2.3 Differentialgleichungen in SAML

In Zukunft sollte man das Sprachkonzept der Zustandsparameter und von Differentialgleichungen voneinander trennen.

Eine verbesserte Möglichkeit der Deklaration von Differentialgleichungen ist in Quelltext 7.1 dargestellt.

Quelltext 7.1: Verbesserte Modellierung von Differentialgleichungen in SAML

```
x1' = 2 init 0;
x2' = x1 +1 init 1;
```

In dieser Deklaration wird auf einen Wertebereich verzichtet, da auch analog beim Lösen der Differentialgleichung keine mathematische Begrenzung des Wertebereich existiert.

7.3 Ausblick

In diesem Abschnitt sollen aufbauend auf der vorherigen Analyse mehrere Vorschläge und Fragestellungen aufgeworfen werden, die sich basierend auf diesen Ergebnissen ergeben.

7.3.1 Fehleranalyse beim numerischen Lösen der Differentialgleichung

Wie bereits in Abschnitt 4.3.1 ausgeführt wurde, verwendet Scilab eine Präzessionsangabe in der numerischen Berechnung der Lösung eines Systems von Differentialgleichungen. Der umgesetzte Prototyp hat deswegen zwar für nicht sichere Systeme wie in Abschnitt 6.2.1 bei dem Experiment der Punktförmigen Zugbeeinflussung als

unsicher einstufen können. Diese Aussage könnte jedoch auch falsch sein, falls der Fehler, der beim Lösen der Differentialgleichung entstanden ist, größer ist als die vorgegebene Präzession, die in dem Prototypen als oberste Fehlergrenze gewählt wurde.

Daraus ergeben sich verschiedene Ansatzpunkte. Zum Einen könnte untersucht werden, wie die Qualität der gewählten Präzession ist und wie sie sich jeweils zum realen lokalen Fehlerwert verhält. Das Ziel könnte sein, eine Güte herauszufinden, mit der ein Gegenbeispiel für ein Erreichbarkeitsproblem wahr ist.

Ein weiterer Ansatzpunkt wäre es, ein numerisches Verfahren innerhalb von z.B. Scilab zu implementieren und dort ähnlich wie in [Nie13] eine Fehlerberechnung durchzuführen. Dies hätte den Nachteil, dass der eigentliche Vorteil, dass man auf die Nutzung einer Standardfunktion zum Lösen von Differentialgleichungen nicht zurückgreifen würde, sondern eigenständig ein numerisches Verfahren implementieren müsste. Man würde jedoch mit dieser Lösung für jeden diskreten Zeitwert einen lokalen Fehler berechnen, den man in dieser Arbeit vorgestellten Transformationsverfahren nutzen könnte.

Darauf aufbauend könnte ein Ansatzpunkt sein, herauszufinden, ob es neben Scilab andere Programme gibt, die bei einem numerischen Lösungsverfahren für Differentialgleichungen für jeden Diskretisierungsschritt einen lokalen Fehler berechnen.

7.3.2 Neuberechnung bei überlappendem Fehlerintervall

Bisher wurde bei einer Fehlerüberlappung in einer Bedingung ein Verfahren zur Anpassung der Übergänge angewandt. Jedoch werden dafür Indeterminismen in das Model eingebaut, die wiederum zur Folge haben, dass mehrere Modelle entstehen, die zuerst transformiert und dann gecheckt werden müssen. Für Transformationen hat sich gezeigt, dass diese (in der momentan Version) bei vielen Diskretisierungsschritten viel zeitlichen Aufwand benötigen. Deswegen könnte es ein Ansatz sein, sofern eine Fehlerüberlappung auftritt, mit einer kleineren Fehlervorgabe eine Neuberechnung des Systems von Differentialgleichungen durchzuführen, um mit einem geringeren Fehlerintervall die Wahrscheinlichkeit auf eine Überschneidung zu reduzieren.

In Algorithmus 10 ist für die Funktion *handleErrorInTransitionConditions* eine veränderte Version angegeben (vgl.: Algorithmus 4).

Algorithm 10 Funktion *handleErrorInTransitionConditions* zur Transformation eines Modell mit aufgelösten Differentialgleichungen

input : The Model H which has to transformed, A Set of k Transitions T, a solved ODE System M
output : The transformed Model H'

```
82 //PART 1
83 C ← getCriticalTransitions(T, M)
84 //PART 2 - Version 2
85 while sizeOf(C) > 1 do
86 |   M = solveODESystemWithChangedError (H)
87 |   C ← getCriticalTransitions(T, M)
```

Der erste Teil ist identisch zu der in Algorithmus 4 beschriebenen Version. Hier werden zuerst alle kritischen Übergänge herausgesucht. Der Ansatz besteht nun daran, das Differentialgleichungssystem mit reduzierter Fehlervorgabe (z.B. Halbierung

des Fehlermaximums err_{max}) erneut zu lösen. Für eine Halbierung des Fehlermaximums err_{max} würde man auch den Fehlerintervall halbieren, was zur Folge hätte, dass der Vergleichswert nicht mehr im Fehlerintervall liegt. Da $\lim_{err_{max} \to 0}[x_i(t_j) - err_{max}, x_i(t_j) + err_{max}] = x_i(t_j)$ gilt, wird in Algorithmus 4 der Fehler solange reduziert, bis kein Übergang im Modell mehr existiert, in dem es eine Überschneidung gibt.

7.3.3 Bounded Model Checking

Im Vergleich mit dem hybriden Model Checker iSAT hat sich gezeigt, dass die Performance stark abhängig ist von dem gewählten t_{end}. Die Wahl von t_{end} wird dem Modellierenden überlassen. Eine Überlegung wäre es nun, dass eine Art von Bounded Model Checking eingesetzt wird. So könnten mehrere Durchgänge des Model Checking Prozess für immer größer werdende t_{end} durchgeführt werden. So könnte z.B. für die Adaptive Cruise Control anfangs eine Grenze von 90 Sekunden festgelegt werden. Anstatt jedoch das Modell für 90 Sekunden zu berechnen, wie es in dem Experiment in Tabelle 6.4 für den „Bad Case“ gemacht wurde, könnte das Modell schrittweise für z.B. erst 10, dann 20 und schließlich 30 Sekunden gecheckt werden. So würde man viele unnötige Schritte vermeiden, die bei einer Grenze von 90 Sekunde nötig sind.

8. Zusammenfassung

In dieser Arbeit sollte ein hybrides Modell in eine diskrete Sprache wie SAML transformiert werden, um anschließend mit einem symbolischen Model Checker eine gestellte Spezifikation an das Modell zu überprüfen.

Dafür wurde ein Transformationsalgorithmus vorgestellt, der zuerst aus dem Modell ein zu lösendes System von Differentialgleichungen extrahiert und dann mit einem Lösungsprogramm für Differentialgleichungen (in dem umgesetzten Prototypen Scilab) numerisch löst. Anschließend wird die gefundene Lösung in das Modell eingefügt. Um dies zu erreichen, wird eine Zeitkomponente verwendet, die endlich viele diskrete Zeitschritte modelliert. Eine Zustandsübergangsbedingung im Modell für einen Wert einer Differentialgleichung wird durch eine Referenz auf die Zeitkomponente ersetzt, wenn zu einem Zeitpunkt t die Zustandsübergangsbedingung wahr ist.

Beim numerischen Lösen der Differentialgleichung entsteht ein Fehler. In dem verwendeten Prototypen wurde angenommen, dass die verwendete Präzision, die eingehalten werden soll, dem Fehler entspricht. Dadurch kann für die Lösung nicht für alle Fälle ausgeschlossen werden, dass die Aussage des Model Checking korrekt ist. Trotzdem kann, wenn eine Lösung gefunden wird, die den Fehler für alle Werte unter einem bzw. mehreren Grenzen hält, durch eine Betrachtung aller Übergänge und anschließender Anpassung der Übergänge das Modell so umgeformt werden, dass ein gefundenes Gegenbeispiel eines Erreichbarkeitsproblems wahr ist.

Ebenso wurde in dieser Arbeit das Verhalten von Differentialgleichungen der Form $\dot{x} = f(x, t, p)$ mit einem Parametervektor p betrachtet. Während der Ausführung eines Modells können sich Werte von p ändern. In diesem Fall wird das Modell so angepasst, dass sobald sich ein Parameter in der Ausführung ein Gegenbeispiel der aktuellen Zustände ausgegeben wird. Ab diesem Zeitpunkt wird eine Neuberechnung und Transformation des Modells durchgeführt.

Anhand von zwei Fallstudien wurde ein implementierter Prototyp getestet, um anhand von diesen Eigenschaften des Verfahrens und des Prototyps festzustellen, um im Anschluss eine Analyse der Eigenschaften durchführen zu können. Es hat sich gezeigt, dass die Performance des Prototyps stark abhängig von der Erstellung der angepassten Zustandsübergänge ist. Bei Modellen, die Differentialgleichungen der Form $\dot{x} = f(x, t, p)$ enthalten, hat sich gezeigt, dass die Performance darunter

leidet, wenn ein oder mehrere Wechsel in der Umgebung der Startzeit t_0 stattfinden, da in diesem Fall einmal oder sogar mehrfach eine neue Transformation des Modells durchgeführt wird, was in diesem Fall bei vielen verbleibenden Diskretisierungsschritten für eine Verschlechterung der Performance sorgt.

Sofern die Wechsel der Parameter jedoch in der direkten Nähe von t_{end} befinden, hat sich gezeigt, dass die Performance bei größeren t_{end} besser ist als der als Vergleich dienende hybride Bounded Model Checker iSAT.

Aus der Arbeit haben sich zwei wesentliche Stellen ergeben, an denen angesetzt werden kann. Zum Einen sollte untersucht werden, wie das Erstellen von Zustandsübergangsbedingung beschleunigt werden kann, um eine starke Performaceverbesserung zu erzielen. Zum Anderen sollte untersucht werden, wie der Fehler beim Lösen einer Differentialgleichung in einem Lösungsprogramm wie Scilab oder Matlab berechnet werden kann oder wie sich die dort festlegbare Präzision auf das Ergebnis auswirkt.

A. Anhang

A.1 SAML Modell der Punktförmigen Zugbeeinflussung

```
t: [[0..6], 0.1];
component train
        /@*ODE: VEL' = t-10*/
        VEL: [0..1] init 50;
        /@*ODE: POS' = VEL*/
        POS: [0..1] init 0;

        alarm : [0..1] init 0;

        VEL >= 25 & POS > 100 -> alarm' = 1;
        VEL < 25 | POS <= 100 -> alarm' = 0;
endcomponent

SPEC !EF (train.alarm = 1);
```

A.2 Tranformiertes Modell der Punktförmigen Zugbeeinflussung

```
//@Generated
component CLOCK
        TIME: [0 .. 60] init 0;
        TIME + 1 < 60 -> TIME' = TIME + 1;
        TIME + 1 >= 60 -> TIME' = TIME;
endcomponent
component train
        alarm: [0 .. 1] init 0;
        (CLOCK.TIME = 0 | CLOCK.TIME = 1 | CLOCK.TIME =
                2 | CLOCK.TIME = 3 | CLOCK.TIME = 4 |
```

```
        CLOCK.TIME = 5 | CLOCK.TIME = 6 | CLOCK.TIME =
        7 | CLOCK.TIME = 8 | CLOCK.TIME = 9 |
        CLOCK.TIME = 10 | CLOCK.TIME = 11 | CLOCK.TIME
        = 12 | CLOCK.TIME = 13 | CLOCK.TIME = 14 |
        CLOCK.TIME = 15 | CLOCK.TIME = 16 | CLOCK.TIME
        = 17 | CLOCK.TIME = 18 | CLOCK.TIME = 19 |
        CLOCK.TIME = 20 | CLOCK.TIME = 21 | CLOCK.TIME
        = 22 | CLOCK.TIME = 23 | CLOCK.TIME = 24 |
        CLOCK.TIME = 25 | CLOCK.TIME = 26 | CLOCK.TIME
        = 27 | CLOCK.TIME = 28 | CLOCK.TIME = 29) &
        (CLOCK.TIME = 27 | CLOCK.TIME = 28 | CLOCK.TIME
        = 29 | CLOCK.TIME = 30 | CLOCK.TIME = 31 |
        CLOCK.TIME = 32 | CLOCK.TIME = 33 | CLOCK.TIME
        = 34 | CLOCK.TIME = 35 | CLOCK.TIME = 36 |
        CLOCK.TIME = 37 | CLOCK.TIME = 38 | CLOCK.TIME
        = 39 | CLOCK.TIME = 40 | CLOCK.TIME = 41 |
        CLOCK.TIME = 42 | CLOCK.TIME = 43 | CLOCK.TIME
        = 44 | CLOCK.TIME = 45 | CLOCK.TIME = 46 |
        CLOCK.TIME = 47 | CLOCK.TIME = 48 | CLOCK.TIME
        = 49 | CLOCK.TIME = 50 | CLOCK.TIME = 51 |
        CLOCK.TIME = 52 | CLOCK.TIME = 53 | CLOCK.TIME
        = 54 | CLOCK.TIME = 55 | CLOCK.TIME = 56 |
        CLOCK.TIME = 57 | CLOCK.TIME = 58 | CLOCK.TIME
        = 59 | CLOCK.TIME = 60) -> (alarm' = 1);
(CLOCK.TIME = 30 | CLOCK.TIME = 31 | CLOCK.TIME
        = 32 | CLOCK.TIME = 33 | CLOCK.TIME = 34 |
        CLOCK.TIME = 35 | CLOCK.TIME = 36 | CLOCK.TIME
        = 37 | CLOCK.TIME = 38 | CLOCK.TIME = 39 |
        CLOCK.TIME = 40 | CLOCK.TIME = 41 | CLOCK.TIME
        = 42 | CLOCK.TIME = 43 | CLOCK.TIME = 44 |
        CLOCK.TIME = 45 | CLOCK.TIME = 46 | CLOCK.TIME
        = 47 | CLOCK.TIME = 48 | CLOCK.TIME = 49 |
        CLOCK.TIME = 50 | CLOCK.TIME = 51 | CLOCK.TIME
        = 52 | CLOCK.TIME = 53 | CLOCK.TIME = 54 |
        CLOCK.TIME = 55 | CLOCK.TIME = 56 | CLOCK.TIME
        = 57 | CLOCK.TIME = 58 | CLOCK.TIME = 59 |
        CLOCK.TIME = 60) | (CLOCK.TIME = 0 | CLOCK.TIME
        = 1 | CLOCK.TIME = 2 | CLOCK.TIME = 3 |
        CLOCK.TIME = 4 | CLOCK.TIME = 5 | CLOCK.TIME =
        6 | CLOCK.TIME = 7 | CLOCK.TIME = 8 |
        CLOCK.TIME = 9 | CLOCK.TIME = 10 | CLOCK.TIME =
        11 | CLOCK.TIME = 12 | CLOCK.TIME = 13 |
        CLOCK.TIME = 14 | CLOCK.TIME = 15 | CLOCK.TIME
        = 16 | CLOCK.TIME = 17 | CLOCK.TIME = 18 |
        CLOCK.TIME = 19 | CLOCK.TIME = 20 | CLOCK.TIME
        = 21 | CLOCK.TIME = 22 | CLOCK.TIME = 23 |
        CLOCK.TIME = 24 | CLOCK.TIME = 25 | CLOCK.TIME
```

```
                    = 26) -> (alarm' = 0);
endcomponent
SPEC !(EF (train.alarm = 1));
```

A.3 iSAT Modell der Punktförmigen Zugbeeinflussung

```
DECL
          define dt = 0.1;
          float [0, 10000] pos;
          float [0, 1000] vel;
          float [-100, 1000] k;
INIT
          pos = 0;
          vel = 50;
          k = 0;
TRANS
          k' = k + 1 * dt;
          pos' = pos + vel * dt;
          vel' = vel + (k - 10) * dt;
TARGET
          pos > 120 and vel >= 25;
```

A.4 SAML Modell der Adaptive Cruise Control

```
t: [[0..60], 0.1];
component carsystem
          /@*ODE: VELCAR1' = ACCCAR1*/
          VELCAR1: [0..1] init 33;
          /@*ODE: POSCAR1' = VELCAR1*/
          POSCAR1: [0..1] init 40;
          /@*ODE-Parameter*/
          ACCCAR1: [-10..1] init 1;

          /@*ODE: VELCAR2' = ACCCAR2*/
          VELCAR2: [0..1] init 33;
          /@*ODE: POSCAR2' = VELCAR2*/
          POSCAR2: [0..1] init 0;
          /@*ODE-Parameter*/
          ACCCAR2: [-10..1] init 1;

          state : [0..1] init 0;

          POSCAR1 - 30 >= POSCAR2 & POSCAR1 > 1000 ->
                    state'=state & ACCCAR2' = 1 & ACCCAR1' = -8;
          POSCAR1 <= 1000 ->
                    state'=state & ACCCAR2' = 1 & ACCCAR1' = 1;
          POSCAR1 - 30 >= POSCAR2 ->
```

```
                state'=0 & ACCCAR2' = ACCCAR2 & ACCCAR1' = ACCCAR1;
        POSCAR1 - 10 >= POSCAR2 & POSCAR1 - 30 < POSCAR2 ->
                state'=0 & ACCCAR2' = -8 & ACCCAR1' = -8;
        POSCAR1 - 10 < POSCAR2 ->
                state'=1 & ACCCAR2' = -8 & ACCCAR1' = -8;
endcomponent

SPEC !EF (carsystem.state = 1);
```

A.5 iSat Modell der Adaptive Cruise Control

```
DECL
        define dt = 0.1;
        define car_size = 10;
        define first_brake = 1000;
        float [0, 10000] pos_car1;
        float [0, 1000] vel_car1;
        float [-10, 10] acc_car1;
        float [0, 10000] pos_car2;
        float [0, 1000] vel_car2;
        float [-10, 10] acc_car2;
INIT
        pos_car1 = 300;
        vel_car1 = 33;
        acc_car1 = 1;

        pos_car2 = 0;
        vel_car2 = 33;
        acc_car2 = 1;
TRANS
                pos_car1' = pos_car1 + vel_car1 * dt;
        vel_car1' = vel_car1 + acc_car1 * dt;
        (pos_car1 > first_brake & vel_car1 > 3) -> (acc_car1' = -8);
        (pos_car1 > first_brake & vel_car1 <= 3)-> (acc_car1' = 0);
        (pos_car1 <= first_brake) -> (acc_car1' = 1);

        pos_car2' = pos_car2 + vel_car2 * dt;
        vel_car2' = vel_car2 + acc_car2 * dt;
        pos_car1 - pos_car2 - car_size < 30 -> acc_car2' = -8;
        pos_car1 - pos_car2 - car_size >= 30 -> acc_car2' = 1;
TARGET
                pos_car1 - pos_car2 - car_size < 0;
```

Literaturverzeichnis

[Att01] Richard Atterer. Design hybrider, eingebetteter systeme - hybride automaten. pages 2–5, 2001. (zitiert auf Seite 8)

[Aul97] Bernd Aulbach. *Gewöhnliche Differentialgleichungen*. Spektrum, Akad. Verlag, 1997. (zitiert auf Seite 5)

[BA12] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012. (zitiert auf Seite 10)

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003. (zitiert auf Seite 12)

[BCM+90] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439. IEEE, 1990. (zitiert auf Seite 12)

[BHWM85] Klemens Burg, Herbert Haf, Friedrich Wille, and Andreas Meister. *Höhere Mathematik für Ingenieure*. Springer, 1985. (zitiert auf Seite 6)

[Bie09] Armin Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–474. IOS Press, 2009. (zitiert auf Seite 12)

[CC97] Earl A Coddington and Robert Carlson. *Linear ordinary differential equations*. SIAM, 1997. (zitiert auf Seite 2)

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. (zitiert auf Seite 12)

[CCN06] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. Modeling and Simulation in Scilab. In *Modeling and Simulation in Scilab/Scicos*, pages 73–100. Springer, 2006. (zitiert auf Seite 29)

[CE82] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982. (zitiert auf Seite 12)

[D+09] Daniel Dvorak et al. Nasa study on flight software complexity. 2009. (zitiert auf Seite 1)

[EH86] E Allen Emerson and Joseph Y Halpern. “sometimes” and “not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986. (zitiert auf Seite 11)

[Enta] Scilab Enterprises. Javasci. Website. Online verfügbar unter http://help.scilab.org/docs/5.5.1/en_US/javasci.html; besucht am 16.Februar 2015. (zitiert auf Seite 29)

[Entb] Scilab Enterprises. ODE. Website. Online verfügbar unter http://help.scilab.org/docs/5.5.1/en_US/ode.html; besucht am 16.Februar 2015. (zitiert auf Seite 29)

[FHT+07] Martin Franzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007. (zitiert auf Seite 12)

[GdSMH01] Anouck Renée Girard, J Borges de Sousa, James A Misener, and J Karl Hedrick. A control architecture for integrated cooperative cruise control and collision warning systems. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 2, pages 1491–1496. IEEE, 2001. (zitiert auf Seite 2)

[Gon13] Tim Gonschorek. Methodik zur Abstraktion kontinuierlicher Modelle. Bachelorarbeit, Universität Magdeburg, Deutschland, June 2013. (zitiert auf Seite 13)

[Gü11] Matthias Güdemann. *Qualitative and Quantitative Formal Model-Based Safety Analysis*. PhD thesis, Unuversität Magdeburg, Germany, 2011. (zitiert auf Seite 11 und 12)

[Hen00] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000. (zitiert auf Seite 9)

[HHWT97] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463. Springer, 1997. (zitiert auf Seite 3)

[iDT10] AVACS H1/2 iSAT Developer Team. isat quick start guide, 2010. (zitiert auf Seite 12)

[Kea96] R Baker Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996. (zitiert auf Seite 13)

[LCJ95] Pamela I Labuhn and William J Chundrlik Jr. Adaptive cruise control, October 3 1995. US Patent 5,454,442. (zitiert auf Seite 1)

[LPN11] Sarah M Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM 2011: Formal Methods*, pages 42–56. Springer, 2011. (zitiert auf Seite 39)

[LSO12a] Michael Lipaczewski, Simon Struck, and Frank Ortmeier. Saml goes eclipse - combining model-based safety analysis and high-level editor support. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI)*, pages 67–72. IEEE, 2012. (zitiert auf Seite 13)

[LSO12b] Michael Lipaczewski, Simon Struck, and Frank Ortmeier. Using tool-supported model based safety analysis - progress and experiences in saml development. In *IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE 2012)*, 2012. (zitiert auf Seite 13)

[MS00] Olaf Müller and Thomas Stauner. Modelling and verification using linear hybrid automata–a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000. (zitiert auf Seite 7 und 8)

[Nie13] Sebastian Nielebock. Komposition gewohnlicher Differentialgleichungen mit sicherheitsrelevanten Zustandsautomaten. Masterarbeit, Universitat Magdeburg, Deutschland, September 2013. (zitiert auf Seite 3, 14, 21, 35, 46, 47, 53 und 56)

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977. (zitiert auf Seite 10)

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982. (zitiert auf Seite 12)

[Sal] G Sallet. Ordinary differential equations with scilab wats lectures provisional notes université de saint-louis 2004. (zitiert auf Seite 30)

[Sch01] Uwe Schöning. Theoretische informatik kurzgefasst. spektrum, 2001. (zitiert auf Seite 6)

[Sch02] Philippe Schnoebelen. The complexity of temporal logic model checking. *Advances in modal logic*, 4(393-436):35, 2002. (zitiert auf Seite 10 und 11)

[SR97] Lawrence F Shampine and Mark W Reichelt. The matlab ode suite. *SIAM journal on scientific computing*, 18(1):1–22, 1997. (zitiert auf Seite 29)

[SWP12] Karl Strehmel, Rüdiger Weiner, and Helmut Podhaisky. *Numerik gewöhnlicher Differentialgleichungen: nichtsteife, steife und differential-algebraische Gleichungen.* Springer Science & Business Media, 2012. (zitiert auf Seite 6)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 18. Mai 2015