UNIKASSEL
VERSITÄT

Fachbereich
Elektrotechnik/Informatik

Fachgebiet
Theoretische Informatik/Formale Methoden

Simon - Ein Simulationstool für Hybride Systeme

MASTERARBEIT

zur Erlangung des Grades eines **Master of Science**
im Fachbereich Elektrotechnik/Informatik
der Universität Kassel

Eingereicht von	Orcun Yörük
Matrikelnummer	31210869
Vorgelegt im	Fachgebiet Theoretische Informatik/Formale Methoden
Gutachter	Prof. Dr. Martin Lange Prof. Dr. Bernhard Sick
Betreuer	MSc Daniel Kernberger
Eingereicht am	06.09.2017

Zusammenfassung

In dieser Masterarbeit geht es um die Simulation von Hybriden Systemen. Hybride Systeme beschreiben sowohl diskretes als auch kontinuierliches Verhalten. Daher sind sie gut geeignet, um zustandsbasierte Szenarien zu modellieren, welche einen Bezug zur *Zeit* haben. Um solche Systeme zu simulieren, wurde im Rahmen dieser Arbeit ein Tool mithilfe der plattformunabhängigen Programmiersprache *Java* entwickelt, welches die gesammelten theoretischen Erkenntnisse in einer praktischen Umsetzung repräsentiert. Das entstandene Tool mit dem Namen *Simon* lässt die korrekte Simulation von Hybriden Systemen zu. Darüber hinaus werden am Ende der Arbeit auch die Grenzen des Tools erläutert und weiterführende Erweiterungs- und Verbesserungsmöglichkeiten aufgezählt.

Die benötigten Grundlagen werden mit Beispielen vermittelt, welche weitestgehend zusammenhängend sind und so den Einstieg in das Thema vereinfachen. Es wird zudem die Lösungsfindung für diverse Problemstellungen, wie dem in solchen Systemen zwangsläufig vorkommenden Nichtdeterminismus, beschrieben und diskutiert.

Diese Masterarbeit ist für Informatik Master-Studenten und Bachelor-Studenten im fortgeschrittenen Studium geeignet, welche sich für die Möglichkeiten der Simulation von Hybriden Systemen interessieren. Eventuell fehlende mathematische Grundlagen werden berücksichtigt und im Grundlagen-Kapitel behandelt.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Ort, Datum Orcun Yörük

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

DiVoX

Im Rahmen des Projekts für das Masterstudium ist die iPad Applikation DiVoX [YY16] entstanden. DiVoX gibt Lehrern die Möglichkeit, Unterrichtseinheiten für Schulexperimente zusammenzustellen, um die Schüler bei der Durchführung digital zu begleiten. Hierfür stehen den Lehrern diverse Module zur Anzeige von Informationen wie Texte, Bilder und Videos zur Verfügung, die in beliebiger Anordnung zusammengesetzt werden können. Darüber hinaus existieren interaktive Module, in denen die Schüler beispielsweise eigene Texte verfassen oder Fragen beantworten können. Auch ist es den Schülern möglich, Bilder und Videos aufzunehmen und diese in das Experiment einzupflegen. Der Lehrer kann im Anschluss auf diese Informationen zugreifen und sie auswerten.

Ein wichtiges Element der grafischen Benutzeroberfläche der App ist das sogenannte V-Diagramm [MM11]. Dabei handelt es sich um ein Lehr- und Diagnoseinstrument, welches bei der Durchführung von biologischen Experimenten eingesetzt werden kann. Es basiert auf dem Prozess der naturwissenschaftlichen Erkenntnisgewinnung und teilt demnach das Experiment in mehrere Phasen (wie beispielsweise Hypothese, Planung und Interpretation) ein, die die Schülern während der Durchführung eines Experimentes bewerkstelligen. Dabei wurde das V-Diagramm zwar speziell für biologische Experimente entworfen, beschränkt sich jedoch nicht auf diese, sodass es auch in anderen naturwissenschaftlichen Experimenten zum Einsatz kommen kann.

Zur Auswertung der Unterrichtseinheiten stehen den Lehrern diverse Tools zur Verfügung. Neben der Möglichkeit die Ergebnisse jedes Schülers einzeln zu bewerten, kann der Lehrer die Antworten aller Schüler zu einer Frage nebeneinander anzeigen lassen, um diese zu vergleichen.

Als Softwarearchitektur kommt bei DiVoX ein Client-Server-Modell zum Einsatz. Dabei dient ein Webserver als zentraler Knotenpunkt, welcher die Daten in einer Datenbank persistiert und diese den Clients über Schnittstellen zur Verfügung stellt. Die iPads mit den installierten Apps sind die Clients, welche mit dem Server über das Internet kommunizieren. Jedes iPad kann als Lehrer- und Schüler-iPad eingesetzt werden. Durch separate Logins gelangen die Anwender entweder zur Lehrer-Ansicht, in der Unterrichtseinheiten definiert und ausgewertet werden können oder zur Schüler-Ansicht, welche die Durchführung der Unterrichtseinheiten ermöglicht.

Idee

Insbesondere unerfahrene Schüler können aufgrund von falschen Annahmen die Experimente schnell zum Scheitern bringen. Um dies im Vorfeld zu verhindern, wäre es nötig, die Schüler umfassend über die falschen und richtigen Ansätze zu unterrichten, damit das Experiment nicht früh scheitert. Dies führt allerdings dazu, dass je nach Experiment unter Umständen zunächst viel erklärt werden müsste. Auf diese Weise wird den Schülern die Möglichkeit genommen, eigenständig den richtigen Lösungsweg zu erarbeiten. Zur Erweiterung und Ergänzung von DiVoX entstand daher die Idee, die Schüler auf die Durchführung der Experimente besser vorzubereiten, indem diese das Experiment vor der realen Durchführung zunächst *simulieren*. So könnten die Schüler durch die Simulation verschiedener Läufe grobe Fehler vermeiden und ein Gefühl dafür bekommen, welche Parameter welche Auswirkung auf den Verlauf des Experiments haben.

Umsetzung

Naturwissenschaftliche Experimente sind oft durch *Differentialgleichungen* beschreibbar. Solche Gleichungen existieren beispielsweise für das mathematische Pendel oder für den Temperatur-Verlauf beim Heizen mit einem Brenner, sodass Differentialgleichungen als mathematische Grundlage dienen können, um Experimente zu formalisieren.

Bei vielen Experimenten reicht eine einzelne Differentialgleichung jedoch oft nicht aus, sodass mehrere Differentialgleichungen benötigt werden. Ein (vereinfachtes) Thermostatventil einer Heizung wird beispielsweise mit zwei Zuständen beschrieben. So kann das Thermostat, abhängig von der Umgebungs- und der eingestellten Temperatur das Ventil öffnen oder schließen. Für dieses System werden zwei Differentialgleichungen benötigt. Dabei beschreibt die erste den Temperaturabfall bei geschlossenem Ventil und die zweite den Temperaturanstieg bei geöffnetem Ventil. Um ein solches Szenario zu modellieren, können *Hybride Systeme* verwendet werden. Solche Systeme beschreiben sowohl diskretes als auch kontinuierliches Verhalten. Damit können Zustandssprünge, wie das Öffnen oder Schließen des Ventils und kontinuierliche Verläufe der Zeit modelliert werden. Hybride Systeme dienen der Simulation daher als theoretische Grundlage.

Die rein theoretische Erarbeitung der Herangehensweise einer Simulation genügt den Anforderungen von Anwendern wie Lehrern und Schülern jedoch nicht. Um die Ergebnisse dieser Arbeit in der Praxis einsetzen zu können, ist daher ein Software Tool sinnvoll, welches die erarbeiteten Lösungen repräsentiert.

1.2 Ziel

Ziel dieser Masterarbeit ist die Entwicklung eines Simulations-Tools, mit dem Lehrer naturwissenschaftliche Experimente beschreiben können und Schüler durch Eingabe diverser Initialwerte bedienen können. Dadurch erhalten die Schüler die Möglichkeit,

unterschiedliche Läufe studieren zu können, um zu entscheiden, welche Initialwerte für das Experiment besser geeignet sind. Das zu entwickelnde Tool sollte den beschriebenen Einsatzzweck bedienen können, jedoch nicht zwangsläufig darauf beschränkt werden. Durch Schnittstellen in der Software sollte das Tool erweiterbar bzw. modifizierbar sein, um weitere Einsatzfelder erschließen zu können. Um das Simulations-Tool auch in einem anderen Umfeld als die iPad App einsetzen zu können, sollte es von dieser abgekapselt werden.

Die Automaten, die das Tool simulieren soll, sind in der Regel *nichtdeterministisch*. Um diesen Nichtdeterminismus in dem Tool behandeln zu können, sollte es dem Anwender möglich sein, verschiedene Entscheidungslogiken zu nutzen. Initial soll das Tool die manuelle Wahl des nächsten Schritts durch den Anwender sowie die zufällige Wahl beherrschen können. Es soll möglich sein, das Programm durch weitere Logiken zu erweitern.

1.3 Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten sowie deren Abgrenzung zu dieser Arbeit beschrieben.

1.3.1 Realzeit-Automaten und UPPAAL

Realzeit-Automaten sind eine Unterklasse von Hybriden Systemen, in denen die kontinuierlichen Variablen fortlaufende Clocks sind. Die Clocks laufen dabei synchron, können jedoch einzeln zurückgesetzt werden. UPPAAL [ABB01] ist ein Tool zur Modellierung, Simulation und Verifikation von Realzeit-Systemen. Im Zusammenhang mit UPPAAL sind zahlreiche Publikationen entstanden. Eine Liste an Publikationen ist unter [UPP15] einzusehen.

Der erste Prototyp des Tools wurde 1993 unter dem Namen TAB entwickelt [BLL97]. Zu Beginn lag der Fokus auf der Verifikation durch Erreichbarkeitsanalysen in Realzeit-Automaten.

Das Tool wurde seitdem sukzessive weiterentwickelt, sodass sich der Funktionsumfang und die Bedienbarkeit verbessert haben und die Berechnungszeiten gesunken sind. Jedoch verwendet UPPAAL noch heute Realzeit-Automaten als Datenmodell. In dieser Arbeit wird versucht, diese Einschränkung aufzuheben, sodass Hybride Systeme im Allgemeinen akzeptiert und als Datenmodell genutzt werden können. In [ABB01] wird die Unterstützung von Hybriden Systemen für die Zukunft zwar angestrebt, jedoch nicht zur Verwendung als Datenmodell, sondern als Basis für die visuelle Animation von Simulationen.

1.3.2 Dynamische Systeme

Zu dynamischen Systemen existieren zahlreiche Publikationen und Bücher. Dynamische Systeme [Gun10] beschreiben Systeme, die sich mit der Zeit verändern. Als Beispiele sind das Wetter, Temperaturverläufe oder Planetensysteme zu betrachten.

Dynamische Systeme sind mathematischer Natur. Ein mathematisches Modell [Kaw17] für dynamische Systeme besteht dabei aus folgenden Elementen:

- Ein Modell der Zeit.
- Die Beschreibung von Zuständen, die das System einnehmen kann.
- Die Beschreibung des dynamischen Gesetztes, nach dem sich das System in der Zeit entwickelt.

Hybride Systeme können daher als ein konkretes mathematisches Modell von dynamischen Systemen betrachtet werden.

Die Möglichkeiten der Simulation dynamischer Systeme wurden vielfach untersucht, so beispielsweise in [Bos13]. Als Werkzeug kommt hierbei allerdings in der Regel die Systemanalyse zum Einsatz. Diese untersucht ein System anhand von Beobachtungen und Feststellungen. So werden unter anderem die Eigenschaften, das Verhalten und die Beziehungen verschiedener Elemente des Systems analysiert und bewertet.

In dieser Arbeit kommen jedoch Differentialgleichungen zum Einsatz, die das System konkret in seiner zeitlichen Entwicklung beschreiben. Daher ist es nicht notwendig, das System zunächst zu beobachten, um dessen Verhalten zu studieren. Des Weiteren werden in dieser Arbeit Hybride Systeme als Model verwendet.

1.3.3 Hybride Systeme

Die Anfänge des Hybriden Systems gehen zurück auf Thomas A. Henzinger, welcher den Begriff der *Hybriden Systeme* eingeführt hat. Dabei beschreibt Henzinger ein Hybrides System als ein dynamisches System mit diskreten und kontinuierlichen Komponenten [Hen96].

Mats Andersson hat 1994 ein Tool namens *OmSim* zur Simulation von Hybriden Systemen entwickelt [And94]. Das Tool kann mit der eigens entwickelten Modellierungssprache *Omola* komplexe Systeme über eine grafische Benutzeroberfläche modellieren und simulieren.

Es folgte 2008 ein umfangreiches Werk mit dem Titel *Hybrid Systems: Modeling, Analysis and Control* [LTS08]. In diesem werden Hybride Systeme intensiv untersucht und Themen wie Erreichbarkeit und Entscheidbarkeit behandelt. Zudem wird das Thema Simulation diskutiert. Dabei werden auch die Problemstellungen bei der Implementierung eines Algorithmus zur Simulation Hybrider Systeme beschrieben. So werden beispielsweise Zeno-Pfade und der Nichtdeterminismus als zu lösende Aufgaben geschildert. Diese werden in Kapitel 3 dieser Arbeit behandelt.

In Abgrenzung zu bereits bekannten Arbeiten zur Simulation Hybrider Systeme und zum beschriebenen Simulationstool OmSim, legt diese Arbeit den Fokus auf den didaktischen Aspekt und die Bedienbarkeit des Programms durch Endnutzer wie Lehrer und Schüler. Zudem stehen naturwissenschaftliche Experimente im Vordergrund.

1.4 Aufbau der Arbeit

In Kapitel 2 werden zunächst die benötigten theoretischen und technischen Grundlagen erarbeitet, welche als *Werkzeuge* in den nachfolgenden Kapiteln benötigt werden. Einen großen Teil nehmen hier Differentialgleichungen und die Lösung solcher ein. Auch werden Hybride Systeme definiert und beschrieben. Es folgt ein kurzer Abschnitt zu den in dieser Arbeit verwendeten technologischen Mitteln.

Im nachfolgenden Kapitel 3 werden Überlegungen zur Simulation beschrieben. Dabei werden zunächst diverse Probleme diskutiert und deren Lösungen vorgestellt. Hier spielt insbesondere der Nichtdeterminismus eine große Rolle. Daraufhin wird der Simulationsverlauf beschrieben, welcher als wichtige Grundlage für die Implementierung von Simon dient.

Kapitel 4 beschreibt daraufhin die Umsetzung von Simon. Zunächst werden die einzelnen Komponenten der Software beschrieben, um anschließend deren Zusammenspiel im Klassendiagramm zu erläutern. Zudem wird die konkrete Umsetzung des Simulationsverlaufs veranschaulicht. In Kapitel 5 geht es um die Vorstellung von Simon. Die Eingabemaske und die Simulationsansicht werden begleitend mit Screenshots ausführlich beschrieben.

Das Kapitel 6 beinhaltet die Abschlussdiskussion. Es werden die erreichten Ziele vorgestellt und mit einem Ausblick weiterführende Ideen erläutert. Abschließend folgt ein Fazit.

2 Grundlagen

Zu Beginn der Grundlagen werden Differentialgleichungen definiert. Anschließend werden die für die Simulation wichtigen numerischen Lösungsverfahren für Differentialgleichungen beschrieben.

Es folgt ein Abschnitt zu den Grundlagen von Hybriden Systemen, welche auf Ebene der Anwendungslogik im Verlauf der Masterarbeit zum Einsatz kommen werden. Abschließend folgt ein Abschnitt mit einem Überblick über das als Leser benötigte technische Hintergrund-Wissen, um der Arbeit folgen zu können.

2.1 Differentialgleichungen

Viele naturwissenschaftliche Experimente werden mithilfe von Differentialgleichungen beschrieben. So z.B. der radioaktive Zerfall oder Beschleunigungsvorgänge. Diese Gleichungen müssen in der Simulation gelöst werden oder es muss eine Annäherung an die Lösung gefunden werden. In diesem Kapitel wird vor allem [GJ09] verwendet. Ergänzend findet [Col81] Verwendung.

2.1.1 Grundbegriffe

Definition 2.1 ***Gewöhnliche Differentialgleichung n-ter Ordnung***

Seien $n \in \mathbb{N}$, $D \subset \mathbb{R}^{n+2}$. Gegeben sei eine Funktion $F\colon D \to \mathbb{R}$. Eine *implizite* gewöhnliche Differentialgleichung n-ter Ordnung hat die Gestalt

$$F(x, y(x), y'(x), \ldots, y^n(x)) = 0\,.$$

Eine Funktion y wird *Lösung* von F auf dem Intervall I genannt, wenn y auf I n-mal stetig differenzierbar ist und $F(x, y(x), y'(x), \ldots, y^n(x)) = 0 \;\; \forall x \in I$.
Der Graph $G := \{(x, y(x))\colon x \in I\}$ heißt *Lösungskurve* von F auf dem Interval I. Liegt eine Gleichung der Form

$$y^{(n)} = f(x, y(x), y'(x), \ldots, y^{(n-1)}(x))$$

vor, so wird von einer *expliziten* Differentialgleichung n-ter gesprochen.

Beispiel 2.2 Wassertank Ausflussgeschwindigkeit

Die Ausflussgeschwindigkeit von Wasser aus einem Wassertank kann mithilfe des *Ausflussgesetzes von Torricelli* als gewöhnliche Differentialgleichung erster Ordnung beschrieben werden. Die Gleichung dazu lautet

$$y'(x) = c \cdot \sqrt{y(x)} \quad \text{mit} \quad c = -\frac{A_2}{A_1}\sqrt{\frac{2g}{1-(\frac{A_2}{A_1})^2}}\,.$$

Dabei ist y die Füllhöhe und x die Zeit. A_1 ist der Flächeninhalt des Querschnitts vom Wassertank und A_2 der Flächeninhalt des Ausflussquerschnitts. Mit g wird die Schwerebeschleunigung, welche auf der Erdoberfläche $g = 9{,}81\ m/s^2$ beträgt, beschrieben. Die Konstante c beschreibt also das Verhältnis zwischen dem Wassertank und des Ausflussquerschnitts mit Berücksichtigung der Schwerebeschleunigung.

Eine gewöhnliche Differentialgleichung kann unendlich viele spezielle Lösungen besitzen. Zur Verdeutlichung wird die Differentialgleichung

$$y' = y$$

betrachtet, welche die allgemeine Lösung

$$y(x) = Ce^x$$

besitzt. Diese ist für beliebiges $C \in \mathbb{R}$ eine spezielle Lösung. Um also eine eindeutige Lösung erhalten zu können, muss eine Konstante C ausgewählt werden. Ist ein Anfangswert gegeben, kann unter dessen Berücksichtigung eine eindeutige Lösung gefunden werden.

Definition 2.3 ***Anfangswertproblem***

Eine gewöhnliche Differentialgleichung n-ter Ordnung zusammen mit den Anfangsbedinungen $y(x_0) = y_0\ ,\ y'(x_0) = y_1\ ,\ \ldots\ ,\ y^{n-1}(x_0) = y_{n-1}$ mit $y_0\ ,\ \ldots\ ,\ y_{n-1} \in \mathbb{R}$ wird *Anfangswertproblem* genannt.

Beispiel 2.4 Wassertank Anfangswertproblem

Beträgt der Flächeninhalt des Wassertanks $1\,m^2$ und die des Ausflussquerschnitts $10\,cm^2$, kann die Konstante $c = -\,0{,}0443$ errechnet werden. Die Differentialgleichung hätte damit die Form

$$y'(x) = -\,0{,}0443 \cdot \sqrt{y(x)}\,.$$

Zusammen mit dem Anfangswert $y(0) = 100\,cm$ liegt ein Anfangswertproblem vor, welches gelöst werden kann.

Definition 2.5 ***Systeme gewöhnlicher Differentialgleichungen erster Ordnung***
Seien $n \in \mathbb{N}$, $D \subset \mathbb{R}^{2n+1}$. Gegeben sei $f : D \to \mathbb{R}^n$. Ein implizites System von gewöhnlichen Differentialgleichungen erster Ordnung ist gegeben durch

$$\begin{aligned} y'(x) &= f(x, y(x), y'(x)) \\ &= f(x, y_1(x), \ldots, y_n(x), y_1'(x), \ldots, y_n'(x)) \,. \end{aligned}$$

Sei $E \subset \mathbb{R}^{n+1}$. Dann heißt

$$y'(x) = F(x, y(x))$$

mit einer Funktion $F : E \to \mathbb{R}^n$ *(explizites) System gewöhnlicher Differentialgleichungen erster Ordnung*. Eine Funktion $y : I \to \mathbb{R}$ heißt Lösung, wenn y einmal stetig differenzierbar ist und $y'(x) = F(x, y(x)) \;\; \forall x \in I$ gilt.

Definition 2.6 ***Anfangswertproblem für Systeme gewöhnlicher Differentialgleichungen erster Ordnung***
Ein System gewöhnlicher Differentialgleichungen erster Ordnung zusammen mit den Anfangsbedingungen

$$y_1(x_0) = y_1^0 \;, \ldots \;, y_1(x_0) = y_n^0$$

oder kurz

$$y(x_0) = y^0 \in \mathbb{R}^n$$

heißt *Anfangswertproblem* für das System.

Satz 2.7 Überführung gewöhnlicher Differentialgleichungen *n*-ter Ordnung
Jede explizite gewöhnliche Differentialgleichung n-ter Ordnung lässt sich in ein äquivalentes System von gewöhnlichen Differentialgleichungen erster Ordnung überführen.

Aufgrund dessen werden in dieser Masterarbeit nur Systeme gewöhnlicher Differentialgleichungen erster Ordnung behandelt. Des Weiteren wird in dieser Arbeit die Unbekannte in der Regel als *Zeit* interpretiert und mit t beschrieben.

Definition 2.8 ***Lipschitz-Stetigkeit [Col81]***
Eine Funktion $f : \mathbb{R}^{\geq 0} \times \mathbb{R}^n \to \mathbb{R}^n$ heißt Lipschitz-stetig, wenn eine Konstante $k > 0$ existiert, sodass für jedes $t \in \mathbb{R}^{\geq 0}, \;\; x_1, x_2 \in \mathbb{R}^n$

$$\| f(t, x_1) - f(t, x_2) \| \leq k \, \| x_1 - x_2 \|$$

gilt.

Satz 2.9 ***Picard-Lindelöf [Met02]***
Eine Funktion $f\colon \mathbb{R}^{\geq 0} \times \mathbb{R}^n \to \mathbb{R}^n$ sei Lipschitz-stetig und stetig.
Dann hat das Anfangswertproblem

$$y'(t) = f(t, y(t))$$

$$y(0) = y_0$$

eine eindeutige Lösung $y\colon [0,\tau] \to \mathbb{R}^n$ für alle $\tau \geq 0$ und alle $y_0 \in \mathbb{R}^n$.

Die bewiesene Existenz einer Lösung sagt allerdings nichts darüber aus, wie einfach diese zu finden ist. Es existieren zwar diverse analytische Lösungsverfahren, um eine eindeutige Lösung zu finden (siehe [GJ09]). Jedoch lassen sich mit diesen nicht immer Lösungen finden. Daher werden für die Simulation in der Regel numerische Lösungsverfahren genutzt, da die genaue Lösungskurve nicht zwangsläufig benötigt wird. Diese werden im nachfolgenden Abschnitt näher erläutert. Beispiel 2.10 zeigt die exakte Berechnung der Lösung für $y(60)$ mit Anfangswert $y(0) = 100$.

Beispiel 2.10 Berechnung einer genauen Lösung
Um das Anfangswertproblem aus Beispiel 2.4 zu lösen, wird zunächst die allgemeine Lösung benötigt. Für die in Beispiel 2.4 verwendete Differentialgleichung

$$y'(t) = -0{,}0443 \cdot \sqrt{y(t)}$$

lautet die allgemeine Lösung

$$y(t) = \tfrac{1}{4}(-0{,}0443 \cdot t + C)^2 .$$

In diese lässt sich der Anfangswert 100 für $x = 0$ einsetzen. Dadurch ergibt sich

$$100 = \tfrac{1}{4}(-0{,}0443 \cdot 0 + C)^2$$

$$100 = \tfrac{1}{4}C^2$$

$$C = \sqrt{400}$$

$$C = 20 .$$

Die errechnete Konstante kann nun verwendet werden, um die spezielle Lösung für $y(60)$ zu erhalten mit

$$y(t) = \tfrac{1}{4}(-0{,}0443 \cdot t + C)^2$$

$$y(60) = \tfrac{1}{4}(-0{,}0443 \cdot 60 + 20)^2$$

$$y(60) = 75{,}186241$$

Somit ist die Lösung für $y(60) = 75{,}186241$.

Bemerkung: Um die Eindeutigkeit einer Lösung des Anfangswertproblems aus Beispiel 2.4 beweisen zu können, kann an dieser Stelle der Satz von Picard-Lindelöf nicht angewandt werden, da die Wurzelfunktion in $y = 0$ nicht Lipschitz-stetig ist. Dennoch lässt sich für diese Differentialgleichung die Existenz einer Lösung beweisen, indem der Satz von Peano [Col81] angewandt wird, welcher eine schwächere Voraussetzung besitzt. Der Satz von Peano trifft zunächst allerdings keine Aussage über die Eindeutigkeit der Lösung. Da die Differentialgleichung aber zusätzlich auch separierbar ist, kann mit einer Trennung der Veränderlichen [Col81] eine Lösung gefunden werden. Diese ist zumindest lokal, in einer Umgebung um $t = 0$, eindeutig, was hier auch ausreichend ist.

2.1.2 Numerische Lösungsverfahren

Ein System von gewöhnlichen Differentialgleichungen erster Ordnung ist gelöst, wenn die gesuchte Funktion $y(t)$ gefunden wird, welche das System in jedem Zeitpunkt $t \in \mathbb{R}$ erfüllt. Jedoch reichen in der Regel numerische Lösungsverfahren aus, die die Lösung approximieren, da das Finden der gesuchten Funktion nicht immer einfach oder gar möglich ist. In der Simulation werden nur die Werte der Funktion zu bestimmten Zeitpunkten benötigt, nicht jedoch die Funktion selbst. Daher werden in dieser Masterarbeit ausschließlich numerische Lösungsverfahren für die Simulation verwendet, da durch moderne Computer eine sehr genaue Approximation mit relativ wenig Rechenaufwand berechnet werden kann.

Exemplarisch werden in dieser Arbeit das vergleichsweise simple Euler-Verfahren sowie das komplexere aber genauere klassische Runge-Kutta-Verfahren näher betrachtet, wofür vor allem [Neu13] sowie [GJ09] verwendet wird. Beide Verfahren basieren auf der gleichen Grundidee, die Lösungsfunktion zu diskreten Zeitpunkten zu approximieren. Die Approximation erfolgt dabei über das sogenannte *Einschrittverfahren*. Im Gegesatz zum *Mehrschrittverfahren* wird hier jeder Zeitpunkt einzeln betrachtet und die vorherigen Zeitpunkte nicht mit in die Approximation hinzugezogen.

Euler-Verfahren

Das Euler-Verfahren basiert auf der Idee, die diskrete Approximation $\tilde{x}$ für eine gegebene Anfangsbedingung zu berechnen. Der Algorithmus dazu sieht wie folgt aus.

Algorithmus 2.11 ***Algorithmus für Euler-Verfahren [Neu13]***

- Zunächst wird eine hinreichend kleine äquidistante Schrittweite $h > 0$ gewählt und der zugehörige Knoten $t_i = t_0 + ih$ definiert.
- Die Werte $y(t_i)$ an den Knoten t_i sind rekursiv definiert durch
  - $y(t_0) = x_0$,
  - $y(t_{i+1}) = y(t_i) + h \cdot f(t_i, y(t_i))$, $\quad i = 0,1,2,\ldots$.

Um die Genauigkeit des Verfahrens zu verbessern, sollte ein möglichst kleines h gewählt werden. Für $h \to 0$ konvergiert die Approximation gegen die tatsächliche Lö-

sung, weshalb ein möglichst kleines h in der Regel eine genauere Lösung liefert. Zu beachten ist jedoch, dass sich der Rechenaufwand dadurch erhöht.
Das Euler-Verfahren ist sehr simpel und lässt sich durch den leicht in Computersystemen umsetzbaren Algorithmus schnell verwenden. Zudem liefert das Euler-Verfahren (bei hinreichend kleiner Schrittweite) brauchbare Schätzwerte.

Beispiel 2.12 Wassertank Euler-Verfahren

Unter Berücksichtigung der errechneten Konstante c und des Anfangswerts aus Beispiel 2.4 ergibt sich folgende rekursive Berechnungsvorschrift:

- $y(0) = 100$,
- $y(t_{i+1}) = y(t_i) + h \cdot \left(-0{,}0443 \cdot \sqrt{y(t_i)}\right), \quad i = 0,1,2,...$.

Soll nun beispielsweise der Wert für $t = 60$, was dem Füllstand nach 60 Sekunden entspricht, berechnet werden, ergeben sich die nachfolgenden Ergebnisse für die entsprechende Schrittweiten-Wahl. Die fehlerhafte Abweichung zum exakten Ergebnis von $y(60) = 75{,}186241$, welcher in Beispiel 2.10 errechnet wurde, wird ebenfalls aufgelistet.

- Schrittweite $h = 60$
  Anzahl durchgeführter Rechnungen: 1
  $y(60) = 73{,}42000\ cm$ *(Abweichung: < 2,350 %)*
- Schrittweite $h = 30$
  Anzahl durchgeführter Rechnungen: 2
  $y(60) = 74{,}33459\ cm$ *(Abweichung: < 1,133 %)*
- Schrittweite $h = 15$
  Anzahl durchgeführter Rechnungen: 4
  $y(60) = 74{,}76801\ cm$ *(Abweichung: < 0,557 %)*
- Schrittweite $h = 1$
  Anzahl durchgeführter Rechnungen: 60
  $y(60) = 75{,}15882\ cm$ *(Abweichung: < 0,037 %)*
- Schrittweite $h = 0{,}01$
  Anzahl durchgeführter Rechnungen: 6.000
  $y(60) = 75{,}18597\ cm$ *(Abweichung: < 0,0004 %)*

Zu erkennen ist, das bereits wenige Zwischenschritte ein deutlich genaueres Ergebnis liefern können. Allerdings erfordert eine sehr kleine Abweichung vom exakten Ergebnis einen vergleichsweise hohen Rechenaufwand. Abbildung 2.1 zeigt ein Balkendiagramm, welches die fehlerhaften Abweichungen beim Euler-Verfahren visualisiert.

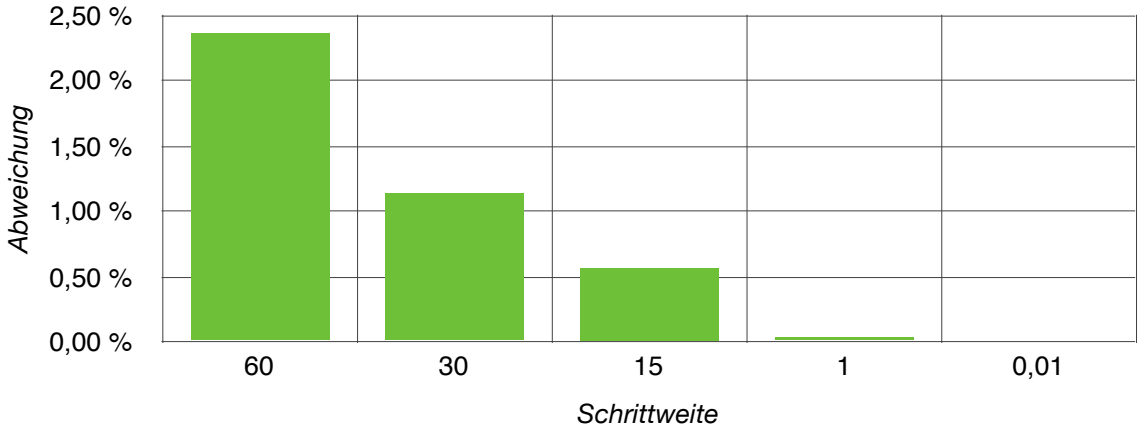

Abbildung 2.1 Abweichung beim Euler-Verfahren

Klassisches Runge-Kutta-Verfahren

Für eine genauere Approximation wird das klassische Runge-Kutta-Verfahren – auch vierstufiges Runge-Kutta-Verfahren genannt – verwendet. Wie das Euler-Verfahren basiert auch das Runge-Kutta-Verfahren auf dem Einschrittverfahren. Jedoch wird beim Runge-Kutta-Verfahren für jeden Zeitpunkt die Richtung mehrfach ermittelt und anschließend ein gewichteter Mittelwert berechnet. Das klassische Runge-Kutta-Verfahren berechnet dabei einmal am Anfang, zwei Mal in der Mitte und einmal am Ende den Wert und berechnet aus diesen Werten den Mittelwert. Dabei werden die Werte der Mitte doppelt gewichtet. Das dabei f für jeden Zeitpunkt viermal ausgewertet muss, macht die Berechnung aufwendiger, als das Euler-Verfahren.

Algorithmus 2.13 ***Algorithmus für klassisches Runge-Kutta-Verfahren [But16] & [Neu13]***

- $k_1 = f(t_i, y(t_i))$,
  $k_2 = f(t_i + \frac{h}{2}, y(t_i) + k_1 \cdot \frac{h}{2})$,
  $k_3 = f(t_i + \frac{h}{2}, y(t_i) + k_2 \cdot \frac{h}{2})$,
  $k_4 = f(t_i + h, y(t_i) + k_3 \cdot h)$,
- $y(t_{i+1}) = y(t_i) + h \cdot \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad i = 0,1,2,\ldots$.

Beispiel 2.14 Wassertank Klassisches Runge-Kutta-Verfahren

Um die Ergebnisse des Euler-Verfahrens mit denen des klassischen Runge-Kutta-Verfahrens vergleichen zu können, wird die gleiche Ausgangssituation aus Beispiel 2.12 gewählt. Auch hier werden die Approximationen mit dem exakten Ergebnis von $y(60) = 75{,}186241$ verglichen. Dabei ergeben sich nach Anwendung des Algorithmus 2.13 die nachfolgenden Ergebnisse:

- Schrittweite $h = 60$
  Anzahl durchgeführter Rechnungen: 1
  $y(60) = 75{,}18652\ cm$ *(Abweichung: < 0,0004 %)*
- Schrittweite $h = 30$
  Anzahl durchgeführter Rechnungen: 2
  $y(60) = 75{,}18626\ cm$ *(Abweichung: < 0,00003 %)*

- Schrittweite $h = 15$
  Anzahl durchgeführter Rechnungen: 4
  $y(60) = 75{,}186242\ cm$ *(Abweichung: < 0,000002 %)*

Es ist ersichtlich, dass das klassische Runge-Kutta-Verfahren bei gleicher Schrittweite an dieser Stelle für eine deutlich genauere Approximation sorgt, als das Euler-Verfahren. Beispielsweise wird hier schon mit der Schrittweite von $h = 60$ ein vergleichbar genaues Ergebnis erreicht, wie beim Euler-Verfahren mit einer Schrittweite von $h = 0{,}01$. In diesem Fall wären also 6000 Zwischenschritte mit dem Euler-Verfahren zur Berechnung von $y(60)$ nötig, um ein vergleichbar genaues Ergebnis wie beim Runge-Kutta-Verfahren zu erreichen.

2.2 Hybride Systeme

Hybride Systeme kommen bei Simon auf Ebene der Anwendungslogik zum Einsatz. Diese werden simuliert und dadurch der aktuelle Zustand der Simulation berechnet und beschrieben.

Ein Hybrides System [Hen96] ist ein System, welches diskretes und kontinuierliches Verhalten beschreibt. Dabei kann das diskrete Verhalten als Zustandsänderung gesehen werden. Das kontinuierliche Verhalten beschreibt Variablen des Systems, die sich über die Zeit verändern. Für dieses Kapitel wird [Yör17] verwendet.

Definition 2.15 ***Hybrides System***

Das Hybride System H ist ein Tupel $H = (L, X, Lab, E, Act, Inv)$ wobei folgendes gilt:

- Die endliche Menge L beinhaltet die diskreten Werte. Die Elemente $l \in L$ heißen *Locations*.
- Die endliche Menge X beinhaltet die kontinuierlichen Variablen. Die Menge von Funktionen $V = \{v \mid v : X \to \mathbb{R}\}$ enthält Wertzuweisungen für diese Variablen. Jede Wertzuweisung $v \in V$ weißt jeder Variablen $x \in X$ einen reellen Wert $v(x) \in \mathbb{R}$ zu.
- Die Funktion $Inv : L \to 2^V$ ordnet jeder Location $l \in L$ eine *Invariante* zu.
- Die Funktion $Act : L \to (X \to DGL_I)$ ordnet jeder Variable $x \in X$ in jeder Location $l \in L$ eine gewöhnliche Differentialgleichung erster Ordnung DGL_I mit globaler Lipschitz-Bedingung auf $\mathbb{R}^{\geq 0}$ zu. Diese werden *Activities* genannt.
- Die endliche Menge Lab beschriftet die Transitionen. Die Elemente $a \in Lab$ heißen *Labels*.
- Die Menge $E \subseteq L \times Lab \times 2^V \times (X \to \mathbb{R}) \times L$ beinhaltet die Übergänge zwischen den Locations. Die Elemente $e \in E$ heißen *Transitionen*. Jede Transition $e = (l, a, G, r, l')$ beinhaltet die Startlocation $l \in L$, die Ziellocation $l' \in L$, das Label $a \in Lab$, den *Guard* $G \in 2^V$ und den *Reset* r, welcher eine partielle Funktion mit $r : X \to \mathbb{R}$ ist.

Bemerkung: In dieser Arbeit wird nur eine Klasse von Hybriden Systemen unterstützt, bei denen die Entwicklung der Variablen über gewöhnliche Differentialgleichungen erster Ordnung beschrieben werden können. Diese Einschränkung wurde getroffen, um die Umsetzung von Simon zu vereinfachen. Allgemein haben Hybride Systeme diese Einschränkung allerdings nicht und können auch andere Gleichungen als Activities nutzen. Darüber hinaus wurde in den vorangegangenen Abschnitten bereits die Theorie für die Simulation von Differentialgleichungen n-ter Ordnung bzw. Systemen von Differentialgleichungen erster Ordnung erarbeitet, sodass Simon dahingehend erweitert werden könnte.

Läufe

Zur Beschreibung der Semantik eines zustandsbasierten Systems werden häufig *Transitionssysteme* verwendet, welche alle möglichen Zustände und Übergänge eines Systems aufzeigen. Da Hybride Systeme unendlich viele Zustände und mögliche Übergänge besitzen, ist ein solches Transitionssystem relativ unübersichtlich. Des Weiteren ist für die Simulation nicht von großer Relevanz, alle möglichen Pfade eines Systems zu kennen. Vielmehr sind einzelne *Pfade*, auch *Läufe* genannt, von Interesse. Fortführende Informationen zu Transitionssystemen von Hybriden Systemen können z.B. [Yör17] entnommen werden.

Der Zustand eines Hybriden Systems ist definiert über dessen aktuelle Location $l \in L$ und der aktuellen Wertzuweisung $v \in V$. Dabei sind die Übergänge zwischen den Locations diskret und die der Variablenwerte kontinuierlich. Hierdurch können *Läufe* beschrieben werden, welche beide Charakteristika abbilden.

Definition 2.16 *Lauf*

Ein *Lauf* eines Hybriden Systems ist eine (endliche oder unendliche) Sequenz

$$q_0 \rightarrow^{t_1} q_1 \rightarrow^{t_2} q_2 \rightarrow^{t_3} \dots$$

mit $q_i = (l_i, v_i) \in L \times V$ und $(t_i)_{i \in \mathbb{N}}$ eine monoton wachsende Folge mit $t_i \in \mathbb{R}^{\geq 0}$, d.h. $0 \leq t_0 \leq t_1 \leq t_2 \leq \dots$. Dabei ist $q_0 = (l_0, v_0)$ der Startzustand mit $v_0 \in Inv(l_0)$. Für jedes q_i wird nun zunächst das Anfangswertproblem $Act(l_i)$ mit Anfangswerten v_i gelöst. Nach Satz 2.9 (Picard-Lindelöf) besitzt dieses auf $[0, t_{i+1} - t_i]$ eine eindeutige Lösung $y : [0, t_{i+1} - t_i] \rightarrow \mathbb{R}^{|x|}$.

Nun gilt entweder:

1. $q_{i+1} = (l_i, v_{i+1})$ mit $v_{i+1} = y(t_{i+1} - t_i)$, oder
2. $q_{i+1} = (l_{i+1}, v_{i+1})$ mit $l_{i+1} \neq l_i$ und es gibt $e \in E$ mit $e = (l_i, a, G_i, r_i, l_{i+1})$, sodass $y(t_{i+1} - t_i) \in G_i$ und $v_{i+1} = v_i|_{r_i}$ mit

$$v_i|_{r_i} = \begin{cases} v_i(x) & \text{wenn } r_i(x) \text{ undefiniert} \\ r_i(x) & \text{sonst} \end{cases}$$

Weiter gilt für alle $t_i \leq t \leq t_{i+1}$, dass $y(t - t_i) \in Inv(l_i)$ und $v_{i+1} \in Inv(l_{i+1})$.

Ein Lauf heißt *zeitkonvergent*, falls es ein $t \in \mathbb{R}^{\geq 0}$ gibt, sodass für alle t_i gilt, dass $t_i \leq t$. Ein *gültiger Lauf* ist ein Lauf, der nicht zeitkonvergent ist.

Beispiel 2.17 Wassertank Hybrides System

In den vorangegangenen Beispielen wurde die Charakteristik der Ausflussgeschwindigkeit eines Wassertanks mithilfe von Differentialgleichungen untersucht. In diesem Beispiel wird ein System, bestehend aus zwei Wassertanks und einem Wasserzulauf, abstrahiert in einem Hybriden System beschrieben. Des Weiteren wird die Dauer, die für das Schwenken des Wasserzulaufs zwischen den beiden Tanks benötigt wird, modelliert. Abbildung 2.2 zeigt schematisch den physikalischen Aufbau des Systems. Dabei sind w_1 und w_2 die beiden Wassertanks. In der Abbildung wird derzeit Wassertank w_1 befüllt. Unten an den Tanks sind Abflüsse zu sehen. Des Weiteren beschreiben h_1 und h_2 die aktuellen Füllhöhen. Mit t wird die Dauer beschrieben, die der Wasserzulauf zum Schwenken zwischen den beiden Tanks zeitlich benötigt.

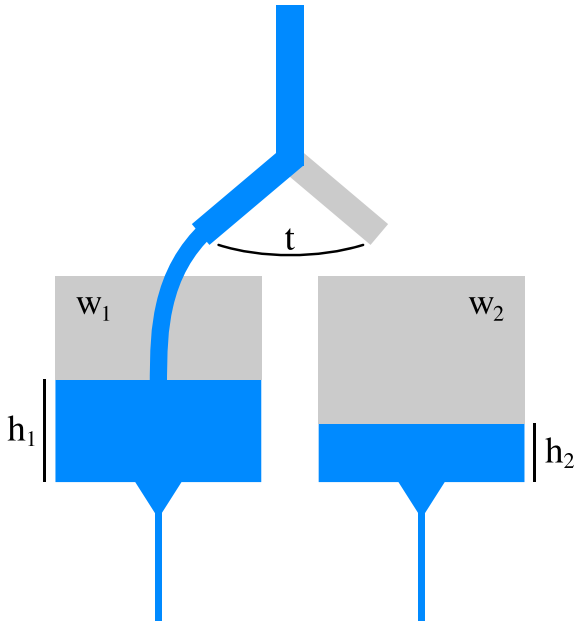

Abbildung 2.2 Wassertank System

Um dieses physikalische Szenario simulieren zu können, muss zunächst dieser Ausschnitt der realen Welt modelliert werden. Dabei finden im Modellierungsprozess diverse Abstraktionen statt. Diese sind nötig weil sonst die Modellierung an sich zu komplex werden würde und damit auch das resultierende Modell. Es ist auch möglich, dass gewisse Aspekte der realen Welt gar nicht erst modelliert werden können, da nicht für jedes Verhalten in der realen Welt auch Gleichungen existieren, die diese formalisieren. Eine mögliche Modellierung des in Abbildung 2.2 gezeigten Aufbaus wird in dem Hybriden System in Abbildung 2.3 gezeigt, welches nachfolgend näher beschrieben wird.
Es existieren vier Locations, welche die diskreten Zustände beschreiben, indem Wassertank w_1 oder Wassertank w_2 befüllt wird oder der Wasserzulauf von w_1 zu w_2 oder von

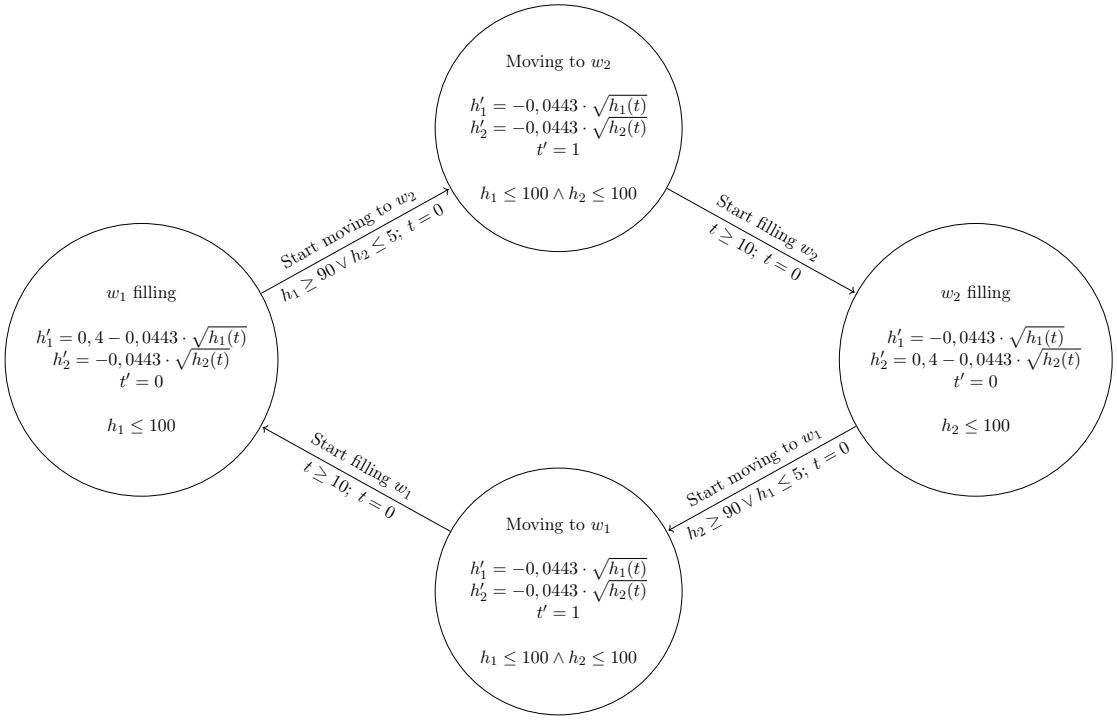

Abbildung 2.3 Wassertank Beispiel

w_2 zu w_1 schwenkt. In jedem Zustand fließt Wasser von beiden Tanks ab. Wenn einer der beiden Tanks befüllt wird, fließt über den Zulauf pro Sekunde 0,4 cm an Füllhöhe in den entsprechenden Tank rein. Der Timer t wird nur in den beiden Locations hochgezählt, indem der Wasserzulauf am Schwenken ist. Des Weiteren sind die Transitionen zwischen den Locations zu erkennen. Anzumerken ist, das in diesem System jede Location nur eine Transition besitzt und so Läufe erzeugt werden, die in ihren diskreten Übergängen periodisch sind. Das kann in anderen Hybriden Systemen anders sein, sodass beispielsweise zwei Locations bidirektional über Transitionen verbunden sein können. Die beiden Locations, in denen einer der Wassertanks befüllt wird, haben als Invariante die maximale Füllhöhe des jeweiligen Wassertanks, welcher hier konkret 100 cm ist. Die Transition zum Schwenken des Wasserzulaufs geschieht frühestens, wenn der aktuell befüllende Wassertank die Füllhöhe von 90 cm erreicht hat oder der andere Wassertank die Untergrenze von 5 cm unterschritten hat. Dabei wechselt das System in den entsprechenden diskreten Zustand, indem der Timer t 10 Sekunden hochzählt und dadurch die Schwenkdauer des Wasserzulaufs simuliert. Es wird (abstrahiert) davon ausgegangen, das während des Schwenkens keiner der beiden Wassertanks befüllt wird. Jede Transition in diesem System hat als Reset das Zurücksetzen des Timers t auf 0.

2.3 Technologischer Hintergrund

2.3.1 Java

Java ist eine objektorientierte, klassenbasierte Mehrzweck-Programmiersprache. Die Sprache ist zwar verwandt mit C und C++ ergänzt diese jedoch mit Ideen aus anderen Programmiersprachen.

Java ist stark typisiert. Das bedeutet, dass Variablen bei der Deklaration einen festen Datentyp erhalten. Dies hat zur Folge, das viele Fehler beim Umgang mit Variablen bereits zum Zeitpunkt der Kompilierung aufgedeckt werden können. Bei Java handelt es sich um eine Hochsprache. Sie abstrahiert die eingesetzte Hardware und setzt sich so von Maschinencode ab. Außerdem wird dem Anwender die Speicherverwaltung mithilfe des *Garbage Collectors* abgenommen, sodass Speicherlecks durch vergessene manuelle Deallokationen vermieden werden können.

Listing 2.1 zeigt ein vollständiges und kompilierbares Beispielprogramm, welches *„Hello World!“* ausgibt.

```
class HelloWorld

  public static void main(String[] args) {
    System.out.println("Hello World!")
  }

}
```

Listing 2.1 Java Beispielprogramm

Durch den hohen Verbreitungsgrad von Java als Programmiersprache wird an dieser Stelle auf eine ausführlichere Beschreibung verzichtet. Bei Bedarf kann unter [GJS00] ein umfangreiches Werk zum Thema Java gefunden werden.

2.3.2 JSON

Die Sprache, welches das zu simulierende Hybride System beschreibt, liegt im JSON Format vor. Simon akzeptiert dieses JSON als Eingabe.

Bei der JavaScript Object Notation – kurz JSON – handelt es sich um ein Textformat zur Serialisierung von strukturierten Daten. JSON ist leichtgewichtig und kann unabhängig von der eingesetzten Programmiersprache verwendet werden. Dieses Kapitel basiert auf [Cro06].

Datentypen

JSON kann vier primitive Datentypen repräsentieren:

- *Strings*: Eine Sequenz aus keinem oder mehreren Unicode Zeichen.
- *Zahlen*: Ganze Zahlen oder Gleitkommazahlen.
- *Boolean*: Ein Datentyp mit nur zwei Zuständen (*true* oder *false*)
- *Null*: Bezeichnung für den Zustand des Fehlens eines Wertes.

Darüberhinaus stehen zwei strukturierte Datentypen zur Verfügung

- *Objekte*: Eine ungeordnete Sammlung von keinen oder mehreren Schlüsselwert-Paaren. Der Schlüssel ist ein String und der Wert ein primitiver oder strukturierter Datentyp.
- *Arrays*: Eine geordnete Folge von keinen oder mehreren Werten. Die Werte können auch hier primitive oder strukturierte Datentypen sein.

Grammatik

Ein JSON Text ist ein serialisiertes Objekt oder Array. Allein primitive Datentypen können kein JSON Text sein und müssen in ein Objekt oder Array gekapselt werden.

Es existieren sechs strukturierende Zeichen:

begin-array	=	[	öffnende eckige Klammer
begin-object	=	{	öffnende geschweifte Klammer
end-array	=	]	schließende eckige Klammer
end-object	=	}	schließende geschweifte Klammer
name-separator	=	:	Doppelpunkt
value-separator	=	,	Komma

Leerzeichen, Returns und horizontale Tabs sind vor und nach den genannten strukturierenden Zeichen erlaubt. Diese haben keine weitere Bedeutung und dienen nur zur besseren Lesbarkeit des JSON Texts für den Menschen.

Ein Wert muss ein Objekt, Array, String oder eine Zahl sein. Darüberhinaus stehen die Literale *true*, *false* und *null* zur Verfügung. Diese Literale müssen kleingeschrieben sein.

value = object / array / string / number / true / false / null

Ein Objekt wird mit öffnenden und schließenden geschweiften Klammern dargestellt und beinhaltet keine oder mehrere Schlüsselwert-Paare.

key-value = string name-separator value

*object = begin-object [key-value *(value-separator key-value)] end-object*

Ein Array wird mit öffnenden und schließenden eckigen Klammern dargestellt und beinhaltet keine oder mehrere Werte. Die Elemente sind Komma-separiert.

*array = begin-array [value *(value-separator value)] end-array*

Eine Zahl kann sowohl als Gleitkommazahl wie auch als ganze Zahl dargestellt werden.

number	=	*[minus] int [frac]*	
frac	=	*decimal-point *digit*	
int	=	*digit *digit*	
digit	=	*0-9*	Ziffer zwischen Null und Neun
minus	=	-	Minus Zeichen
minus	=	*0*	Zahl Null

Ein String wird mit Anführungszeichen umschlossen und beinhaltet keine oder mehrere Unicode Zeichen.

string = *quotation-mark* **char* *quotation-mark*
char = Ein Unicode-Zeichen
quotation-mark = " Anführungszeichen

Parser

Ein JSON Parser wandelt einen JSON Text in eine andere Representation um. Dabei handelt es sich oft um eine Verkettung von *Arrays* und *Dictionarys* in der jeweiligen Programmiersprache. In einigen Fällen wird auch eine eigene JSON Modellstruktur ausgegeben. Der Vorteil in der geparsten Representation liegt darin, dass sich diese in der verwendeten Programmiersprache besser handhaben lässt. In vielen Programmiersprachen sind bereits JSON Parser integriert. Des Weiteren werden zudem viele verbesserte Varianten (oder solche für bestimmte Einsatzzwecke) von Entwicklern zur freien Nutzung bereitgestellt.
Ebenso verhält es sich mit JSON Generatoren. Diese wandeln eine bestimmte Representation, beispielsweise ein Datenmodell, in einen JSON Text um.

Beispiele

Listing 2.2 zeigt ein JSON Objekt, welches eine Person beschreibt.

```
{
   "name":"Max Mustermann",
   "place":"Kassel",
   "age":27,
   "hobbies":["Climbing", "Swimming"]
}
```

Listing 2.2 JSON Objekt

Listing 2.3 zeigt ein JSON Array, welches zwei JSON Objekte beinhaltet, die jeweils einen Standort beschreiben.

```
[
   {
      "latitude":51.312711,
      "longitude":9.479746,
      "city":"Kassel",
      "country":"Deutschland"
   },
   {
      "latitude":37.774929,
      "longitude":-122.419416,
      "city":"San Francisco",
      "country":"USA"
   }
]
```

Listing 2.3 JSON Array

3 Simulation

Der Begriff *Simulation* ist breit gefächert und findet in unterschiedlichen Gebieten Anwendung. Im Kontext dieser Masterarbeit geht es konkret um die Simulation theoretischer Modelle zur Analyse von realen Systemen. Der zu simulierende Ausschnitt der realen Welt muss daher zunächst als theoretisches Modell vorliegen. Als Modell werden in dieser Arbeit Hybride Systeme verwendet. Konkret werden dabei die Läufe von Hybriden Systemen simuliert. Gültige Läufe können beliebig lange simuliert werden. Ist der Lauf hingegen an einem Zeitpunkt ungültig, bricht auch die Simulation ab.

In diesem Kapitel werden die generelle Herangehensweise und die Lösung diverser Problemstellungen im Zusammenhang mit der Simulation von Hybriden Systemen diskutiert. Des Weiteren wird der Ablauf einer Simulation skizziert. Es finden Teile aus [Har96] Verwendung.

3.1 Nichtdeterminismus als Problem

Die Simulation in dieser Arbeit imitiert den Ablauf eines realen, kontinuierlichen Systems, dessen Zustand sich über die Zeit verändert. Ein großes Problem bei solchen Systemen besteht in der kontinuierlichen Zeit, genauer im *kontinuierlichen Nichtdeterminismus*. Nichtdeterminismus im Allgemeinen beschreibt den Umstand, dass sich ein System bei der gleichen Eingabe unterschiedlich Verhalten kann. Es existieren also mehrere Möglichkeiten für Übergänge in den Folgezustand. Ein solches Verhalten ist von theoretischer Natur und muss in der konkreten Umsetzung der Simulation aufgelöst werden, sodass das Tool weiß, wie es sich zu verhalten hat.

Zwischen zwei beliebigen Zeitpunkten existieren in einem Hybriden System unendlich viele Zustände. Daraus folgt, das Transitionen nicht zu einem bestimmten Zeitpunkt, sondern während eines gewissen Zeitintervalls mit unendlich vielen Zwischenschritten möglich sind. Es ist nicht klar, zu welchem Zeitpunkt die Transition gewählt werden soll. Genau hier liegt der kontinuierliche Nichtdeterminismus, welcher sich in einer Computersimulation nicht ohne Weiteres umsetzen lässt.

Mit dem *diskreten Nichtdeterminismus* existiert eine weitere Problemstellung, die behandelt werden muss. Dieser liegt vor, wenn zu einem Zeitpunkt eine oder mehrere Transitionen gewählt werden können. Zu den möglichen nächsten Schritten zählt neben der Wahl eines der möglichen Transitionen auch der Verbleib in der aktuellen Location. Sollten also beispielsweise zwei mögliche Transitionen existieren und der Verbleib in

der aktuellen Location valide sein, gäbe es drei Möglichkeiten, von denen eine gewählt werden muss.

3.2 Vorgehen bei Nichtdeterminismus

Um den Nichtdeterminismus aufzulösen, wird in dieser Masterarbeit der Ansatz gewählt, die Zeit zu diskretisieren und die Simulation in äquidistanten Schritten zu berechnen. Alternativ bestünde noch die Möglichkeit, die äquidistanten Schritte aufzulösen und ungleiche Schrittweiten einzusetzen. Denkbar wäre beispielsweise ein Herantasten an den nächsten Zeitpunkt, an dem eine Transition stattfindet. Möglich wäre das, indem die Lösungskurve anhand mehrerer Punkte analysiert und so der gewünschte Zeitpunkt herausgefunden werden kann. Denkbar wäre ebenfalls, die Differentialgleichungen nach den gewünschten Werten aufzulösen. Beide Lösungsansätze sind allerdings nicht trivial. Außerdem würde so in den natürlichen Verlauf der Simulation eingegriffen werden, indem die Simulation gesteuert werden würde und der Verlauf damit eine Gewichtung bekommt.

Durch die Diskretisierung der Zeit entstehen Seiteneffekte. Positiv ist, dass zeitkonvergente Pfade wegfallen. Solche Pfade können in Systemen vorkommen, indem der zeitliche Abstand zwischen zwei aufeinanderfolgenden Zuständen immer kleiner wird und die Zeit so konvergiert. Durch die diskreten und äquidistanten Schritte fallen diese unrealistischen Pfade automatisch raus.

Durch die Diskretisierung entsteht allerdings ein anderes Problem: Die *Zeitgranularität*. Wird eine zu große Schrittweite gewählt, kann es passieren, dass in der Simulation das Intervall übersprungen wird, indem eine Transition gewählt werden sollte. Beispiel 3.1 verdeutlicht das Problem und beschreibt zugleich die Problematik bei der Wahl einer geeigneten Schrittweite.

Beispiel 3.1 Zeitgranularität

Das folgende Hybride System beschreibt das Modell einer stark vereinfachten Baustellenampel, bei der eine Fahrbahn gesperrt ist und somit die übrige Fahrbahn geteilt werden muss. Das System besteht aus zwei Locations A und B, welche die Durchfahrt für je eine der beiden Fahrbahnen erlauben und für die andere blockieren. Dabei existiert der Timer t, der die Zeit hochzählt. Der Wechsel zwischen den beiden Locations ist frühestens nach 60 Sekunden möglich und muss spätestens nach 80 Sekunden erfolgen, um die Grünphasen fair zu verteilen. Abbildung 3.1 zeigt das zugehörige Hybride System.

Wird nun beispielsweise die Schrittweite 10 gewählt, würde die Simulation in 10 Sekunden-Ticks erfolgen. Es wird also in 10 Sekunden Abständen geschaut, ob die Invarianten valide sind und ob gültige Transitionen vorliegen und bei Bedarf eine Transition gewählt.

Wird allerdings die Schrittweite 50 gewählt, wird nach dem ersten Tick festgestellt, dass die Invarianten erfüllt sind und keine Transition gewählt werden kann. Im zweiten

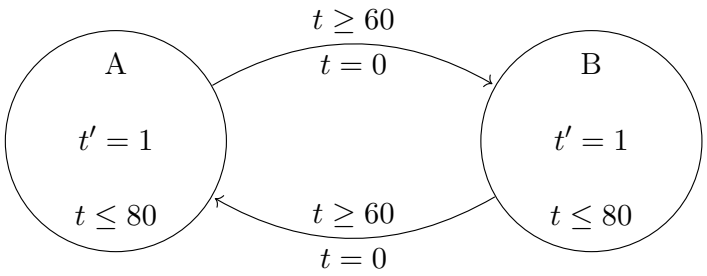

Abbildung 3.1 Baustellenampel Beispiel

Tick, hat der Timer bis 100 Sekunden gezählt und dadurch ist die Invariante in der aktuellen Location nicht erfüllt. Das System hätte früher zur anderen Location wechseln und durch den Reset den Timer zurücksetzen müssen, um auf einem gültigen Pfad zu bleiben. Das gültige Intervall zwischen 60 und 80 Sekunden wurde durch die zu große Schrittweite übersprungen.

Ein weiteres Beispiel wäre die Schrittweite 35. Hier ist es nur an einem einzigen Tick möglich, die Transition zur anderen Location zu gehen, nämlich im zweiten Tick, wenn der Timer bei 70 Sekunden ist. Obwohl in der Theorie unendlich viele Zeitpunkte innerhalb eines Intervalls zum Übergang existieren, bleibt durch die Diskretisierung in Kombination mit der Schrittweite 35 eine einzige Möglichkeit übrig. Darüber hinaus würde in der Simulation die Dauer einer Grünphase immer bei 70 Sekunden liegen, obwohl im Hybriden System Grünphasen mit 60 bis 80 Sekunden erlaubt wären. Die Schrittweite kann also bei der Genauigkeit des Simulationsverlaufs eine Rolle spielen.

Dadurch, dass das System nur noch zu diskreten Zeitpunkten betrachtet und analysiert wird, kann das System auf diese Weise in einen undefinierten Zustand gelangen. Dies ist der Fall, wenn die Invariante der aktuellen Location nicht (mehr) erfüllt ist. Nicht diskretisiert existiert ein Intervall, indem eine Transition möglich ist. Diskretisiert sollte vorzugsweise wenigstens ein Zeitpunkt getroffen werden, indem die Transition möglich ist. Dies wird umso schwerer, wenn das Intervall sehr kurz ist. Um dieses Problem zu lösen, sollte die Schrittweite entsprechend klein gewählt werden. Dies ist jedoch abhängig vom gewählten Hybriden System, sodass diese Entscheidung dem Anwender von Simon überlassen wird. Eine kleinere Schrittweite ist in der Regel genauer und näher an der realen Welt, bei entsprechend erhöhten Rechenaufwand. Allerdings bleibt auch bei einer sehr kleinen Schrittweite eine gewisse Unsicherheit in den Intervallen zwischen den diskreten Zeitpunkten. Denn die Überprüfung der Invarianten und Guards sowie die Handhabung von Transitionen sind nur noch an bestimmten Zeitpunkten möglich.

Der diskrete Nichtdeterminismus wird aufgelöst, indem an jedem diskreten Zeitschritt ein zufälliger Weg gewählt wird. Dies erfolgt über eine gleichverteilte Wahl

des nächsten Schrittes über alle Möglichkeiten hinweg. Zu den möglichen nächsten Schritten zählt die Wahl eines der möglichen Transitionen oder der Verbleib in der aktuellen Location.

Eine gewichtete Verteilung ist abhängig vom eingesetzten Hybriden System, sodass diese zwar optional vom Tool angeboten werden könnte, jedoch nur mit der Möglichkeit, dass der Nutzer die Verteilung definieren kann. Aus Komplexitätsgründen wurde auf diese Erweiterung verzichtet. Jedoch kann eine solche Erweiterung für bestimmte Systeme von Vorteil sein.

Des Weiteren wird dem Anwender die Möglichkeit gewährt, die zufällig gewählte Transition manuell zu verändern. So kann der Anwender gezielter eingreifen und Verhaltensmuster nachspielen.

Seiteneffekte

Durch die getroffenen Maßnahmen entsteht allerdings ein weiterer, unter Umständen massiver Seiteneffekt in Zusammenhang mit der gewählten Schrittweite. In der Regel ist eine kleinere Schrittweite genauer und liefert einen an der Realität näheren Verlauf der Simulation ab. Jedoch kann die gewählte Schrittweite die Wahl der Transition aufgrund der gleichverteilten zufälligen Wahl des nächsten Schritts beeinflussen. Existieren beispielsweise zwei Transitionen, wovon eine gewählt werden kann und die andere erst zu einem späteren Zeitpunkt, hängt es von der Schrittweite ab, wie oft die valide Transition zur Auswahl steht. Damit beeinflusst die Schrittweite das Verhalten der Simulation maßgeblich. Dabei kann eine kleinere Schrittweite, die eigentlich ein besseres Ergebnis liefern sollte, die Situation intensivieren. Beispiel 3.2 veranschaulicht das beschriebene Phänomen.

Beispiel 3.2 Diskretisierung und Schrittweite

Abbildung 3.2 zeigt ein Hybrides System, welches das beschriebene Phänomen verdeutlicht.

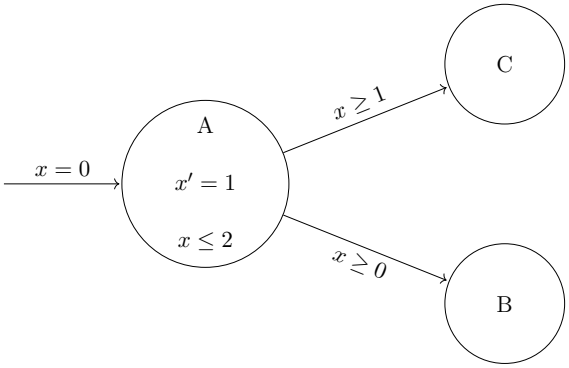

Abbildung 3.2 Hybrides System mit Schrittweiten-Phänomen

Wird für die Simulation des Hybriden Systems in Abbildung 3.2 die Schrittweite 1 gewählt, stehen im ersten Tick der Verbleib in Location *A* und die Transition zu Loca-

tion B zur Auswahl. Die Transition zu Location C ist noch nicht möglich, da der Guard $x \geq 1$ mit $x = 0$ noch nicht erfüllt ist. Entscheidet die Simulation in Location A zu verweilen, steht im zweiten Tick auch die Transition zu Location C zur Auswahl. Die Wahrscheinlichkeit, dass Location C besucht wird, bevor die Transition zu Location B gewählt wurde, liegt somit bei $\frac{1}{2}$.

Wird allerdings als Schrittweite 0,5 gewählt, steht die Transition zu Location C erst im dritten Tick zur Verfügung. Somit sinkt die Wahrscheinlichkeit auf $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, da die Simulation zwei Mal hintereinander den Verbleib in Location A wählen muss. Mit Schrittweite 0,1 liegt die Wahrscheinlichkeit nur noch bei $\frac{1}{1024}$ und somit bei $< 0{,}1$ %, da die Simulation hier zehn Mal hintereinander die Transition zu Location B ablehnen müsste. In diesem Szenario hat die gewählte Schrittweite also einen großen Einfluss auf den Verlauf der Simulation.

3.3 Alternativer Ansatz

Die in Abschnitt 3.2 beschriebenen Schritte werden benötigt, um Hybride Systeme simulieren zu können. Jedoch erfordern diese gewisse Einschränkungen und rufen unter Umständen unerwünschte Effekte hervor. In diesem Abschnitt wird eine mögliche Alternative vorgestellt und dessen Vor- und Nachteile diskutiert.

Eine mögliche Alternative wäre, den zeitlichen Nichtdeterminismus von der Wahl der Transitionen zu trennen. So würde die Abhängigkeit von der Schrittweite wegfallen. Dazu könnte beispielsweise

1. zunächst mit einer Wahrscheinlichkeitsverteilung bestimmt werden, wie lange in der aktuellen Location verblieben werden soll und
2. anschließend an den bestimmten Zeitpunkt eine mögliche Transition nichtdeterministisch gewählt werden.

Hier stellt sich jedoch die Frage wie die Wahrscheinlichkeitsverteilung zu wählen ist, da diese abhängig von den vorliegenden Guards, Invarianten und Differentialgleichungen ist. Es muss die Gefahr ausgeschlossen werden, zu schnell in einen undefinierten Zustand zu gelangen. Dennoch darf die Simulation nicht zu starr arbeiten, damit es möglichst realitätsnah bleibt. Des Weiteren wären die erwähnten zeitkonvergenten Pfade mit dieser Alternative wieder möglich, sodass im Allgemeinen diese Lösung ebenfalls nicht trivial und einschränkungsfrei wäre.

3.4 Ablauf einer Simulation

Wie in Abschnitt 3.2 bereits erwähnt, wird der kontinuierliche Verlauf des Hybriden Systems diskretisiert. Daraus ergeben sich einzelne *Simulationsschritte* die verkettet den Verlauf der Simulation ergeben.

Simulationsschritt

Am Anfang eines Simulationsschritts wird die Invariante der aktuellen Location geprüft. Findet eine Transition zu einer anderen Location statt, wird auch deren Invariante geprüft. Sollte die Invariante einmal nicht erfüllt sein, muss die Simulation beendet werden, da ein ungültiger Lauf vorliegt und das Verhalten nicht spezifiziert ist.

Sollte eine Transition in eine andere Location erfolgen, wird nach dem Wechsel zur neuen Location der Reset der Transition ausgeführt. Anschließend – bzw. direkt, wenn keine Transition stattfand – wird die Activity $Act(l_i)$ der aktuellen Location berechnet. Die Activities werden in dieser Arbeit durch gewöhnliche Differentialgleichungen erster Ordnung beschrieben, welche für den gegebenen Zeitpunkt t_i mit gegebenen Werten $v(t_i)$ mit denen in eine Location gewechselt wird, ein Anfangswertproblem beschreiben, das nach dem Satz von Picard-Lindelöf eine eindeutige Lösung besitzt. Da das explizite Lösen nicht immer möglich ist, werden hier numerische Verfahren verwendet, die die tatsächliche Lösung approximieren. Diese numerischen Verfahren berechnen Approximationen an den zuvor mit Schrittweite h diskretisierten Zeitpunkten. Die Genauigkeit hängt dabei von der Schrittweite h und dem gewählten numerischen Verfahren ab. Wie in Abschnitt 2.1.2 gezeigt wurde, ist das klassische Runge-Kutta-Verfahren dem Euler-Verfahren in der Genauigkeit überlegen. Im Anschluss wird geprüft, welche Transitionen zur Option stehen und bei Bedarf eine der Transitionen ausgewählt oder in der aktuellen Location verweilt. Die Auswahl geschieht gleichverteilt zufällig über alle Möglichkeiten.

Beispiel 3.3 Wassertank Hybrides System Lauf

Im Folgenden wird beispielhaft ein Lauf am Hybriden System, welches im Beispiel 2.17 vorgestellt wurde, durchgespielt. Dies soll dazu dienen, die Idee eines Simulationsschritts zu verdeutlichen und eine Vorstellung für den gesamten Ablauf der Simulation zu bekommen. Bevor die Simulation starten kann müssen anwenderseitig die Initialwerte und die Startlocation festgelegt werden. Des Weiteren muss die Schrittweite h angegeben werden.

- Aktuelle Füllhöhe der Wassertanks: $h_1 = 20\ ;\ h_2 = 20$
- Timer: $t = 0$
- Startlocation: $w_1\, filling$
- Schrittweite: $h = 1$

Tick 0 → 1

- Aktuelle Wertebelegung: $h_1 = 20\ ;\ h_2 = 20\ ;\ t = 0$
- Invariante der aktuellen Location checken: $h_1 < 100 \rightarrow$ OK
- Transition findet nicht statt
- Berechnung der Activities durch Näherungsverfahren *(hier nach Euler)*
- Neue Wertebelegung: $h_1 = 20{,}20188\ ;\ h_2 = 19{,}80188\ ;\ t = 0$
- Mögliche Transitionen checken: Keine Transitionen möglich

Tick 1 → 100

- *Abgekürzt, da lediglich Änderung der Wertebelegung ohne Transitionen*

Tick 100 → 101

- Aktuelle Wertebelegung: $h_1 = 36{,}36318\ ;\ h_2 = 5{,}06037\ ;\ t = 0$
- Invariante der aktuellen Location checken: $h_1 < 100 \rightarrow$ OK
- Transition findet nicht statt
- Berechnung der Activities durch Näherungsverfahren *(hier nach Euler)*
- Neue Wertebelegung: $h_1 = 36{,}49604\ ;\ h_2 = 4{,}96071\ ;\ t = 0$
- Mögliche Transitionen checken:
  - Transition zu *Move to* w_2 möglich → Transition gewählt

Tick 101 → 102

- Aktuelle Wertebelegung: $h_1 = 36{,}49604\ ;\ h_2 = 4{,}96071\ ;\ t = 0$
- Invariante der aktuellen Location checken: $h_1 < 100 \rightarrow$ OK
- Transition findet statt:
  - Wähle neue Location *Moving to* w_2 als aktuelle Location aus
  - Führe Resets der Transition *Move to* w_2 aus: $t = 0$
- Berechnung der Activities durch Näherungsverfahren *(hier nach Euler)*
- Neue Wertebelegung: $h_1 = 36{,}22842\ ;\ h_2 = 4{,}86204\ ;\ t = 1$
- Mögliche Transitionen checken: Keine Transitionen möglich

Tick 102 → 110

- *Abgekürzt, da lediglich Änderung der Wertebelegung ohne Transitionen*

Tick 110 → 111

- Aktuelle Wertebelegung: $h_1 = 34{,}12281\ ;\ h_2 = 4{,}10821\ ;\ t = 9$
- Invariante der aktuellen Location checken: *true* → OK
- Transition findet nicht statt
- Berechnung der Activities durch Näherungsverfahren *(hier nach Euler)*
- Neue Wertebelegung: $h_1 = 33{,}86403\ ;\ h_2 = 4{,}01842\ ;\ t = 10$
- Mögliche Transitionen checken:
  - Transition zu *Fill* w_2 möglich → Transition gewählt

Tick 111 → 112

- Aktuelle Wertebelegung: $h_1 = 33{,}86403\ ;\ h_2 = 4{,}01842\ ;\ t = 10$
- Invariante der aktuellen Location checken: *true* → OK
- Transition findet statt:
  - Wähle neue Location *Filling* w_2 als aktuelle Location aus
  - Führe Resets der Transition *Fill* w_2 aus: $t = 0$
- Berechnung der Activities durch Näherungsverfahren *(hier nach Euler)*

- Neue Wertebelegung: $h_1 = 33{,}60624$; $h_2 = 4{,}32961$; $t = 0$
- Mögliche Transitionen checken: Keine Transitionen möglich

Ein grafischer Verlauf der Simulation ist dem Screenshot in Abbildung 3.3 zu entnehmen. Dieser zeigt bereits die Simulationsansicht von Simon. Das Tool selbst wird in Kapitel 4 und 5 ausführlich vorgestellt.

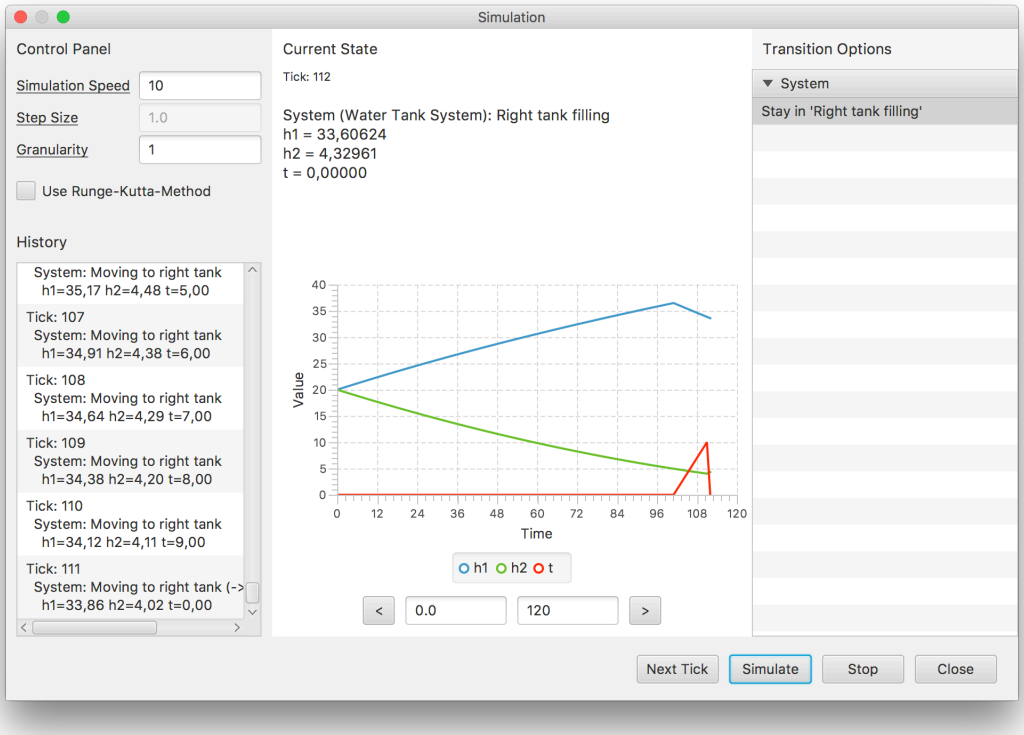

Abbildung 3.3 Screenshot: Simulation Wassertank

4 Umsetzung

In diesem Kapitel wird zunächst der interne Aufbau von Simon zur Simulation Hybrider Systeme und die umgesetzte Kernlogik vorgestellt und diskutiert. Dabei wird insbesondere auf die technische Umsetzung und die Funktionsweise des Programms eingegangen.

4.1 Komponenten

Simon besteht aus mehreren Komponenten, welche jeweils einen gewissen Aufgabenbereich abdecken und untereinander in Beziehung stehen.

Die grafische Benutzeroberfläche

Die *grafische Benutzeroberfläche* ist die Schnittstelle zwischen Programm und Anwender. Sie bietet mit Aktionselementen wie *Buttons* oder *Textfeldern* dem Anwender die Möglichkeit, mit dem System zu interagieren. Zudem erlaubt sie dem Anwender, mit Informationselementen wie *Tabellen* oder *Labels*, den aktuellen Zustand des Tools einzusehen. Die verschiedenen Ebenen der grafischen Benutzeroberfläche werden in Kapitel 5 im Detail vorgestellt.

Dateiformat

Im Tool besteht für den Anwender die Möglichkeit, die getätigten Eingaben in Dateien abzuspeichern. Dabei werden Textdateien gespeichert, welche die Daten im JSON Format beinhalten. Hierfür werden zwei Dateien benötigt:

- Die Definitions-Datei, welche die Hybriden Systeme beinhaltet. Diese Datei hat die Endung *.simd*.
- Die Konfigurations-Datei, welche konkrete Instanzen mit Initialwerten beinhaltet. Daher gehört eine Konfigurations-Datei immer zu einer bestimmten Definitions-Datei. Diese Datei hat die Endung *.simc*.

Das JSON Format, welches in den Dateien zum Einsatz kommt, wird in [Yör17] beschrieben. Dieses Format dient ebenso als Grundlage für den nachfolgend beschriebenen Parser.

Der Parser

Öffnet der Anwender ein vorhandenes Paar von *.simd* und *.simc* Dateien, parst der *Parser* diese in ein Datenmodell, welches in der Eingabemaske Verwendung findet. Ohne diesen Parser müssten die Eingaben des Benutzers direkt in die JSON Datei geschrieben und von dort bei jedem Zugriff wieder ausgelesen werden. Zum Einen wäre so die Zugriffszeit auf die benötigten Daten höher und zum Anderen würde auf diese Weise fehleranfälliger und undurchsichtiger Programmcode entstehen.

Mit dem geparsten Datenmodell wird die Eingabe des Anwenders abstrahiert und strukturiert. Des Weiteren können benötigte Anpassungen, aufgrund von Änderungen in der Struktur der JSON Datei, effizienter eingepflegt werden. Zudem wird durch das Datenmodell das Fehlerhandling und die anschließende Verarbeitung durch den *Modell Mapper* vereinfacht.

Der Modell Mapper

Das Parser Datenmodell ist dazu geeignet, die Benutzereingaben auf der Eingabemaske entgegenzunehmen und zu validieren. Jedoch ist das Modell nicht gut geeignet, um anschließend die Simulation darauf auszuführen. Das Datenmodell ist darauf ausgelegt, die JSON Dateien zu repräsentieren und deshalb fehlt dem Modell die Struktur eines Hybriden Systems.

Der *Modell Mapper* nimmt als Eingabe das Parser Modell und mapped dieses in ein programminternes Hybrides System. Dieses Hybride System wird intern *AutomatonSystem* genannt und dient als Grundlage für die Simulation. Das Mapping erfolgt beim Übergang von der Eingabemaske zur Simulationsansicht.

Das AutomatonSystem

Das *AutomatonSystem* ist der Kern der App. Im Wesentlichen repräsentiert es als eine eigenständige Klasse programmintern das Hybride System. Die Klasse bietet diverse Schnittstellen, die es dem Tool ermöglichen, auf das Hybride System zuzugreifen und die Simulation auf ihr durchzuführen. Dabei kann auch der aktuelle Zustand des Systems ausgelesen werden.

Programmbibliotheken

net.objecthunter.exp4j

Zur Berechnung von mathematischen Ausdrücken wird die Bibliothek *exp4j* eingesetzt. Sie bietet eine komfortable Schnittstelle mit Variablenhandling, welche durch die in der Eingabemaske definierten Properties bequem zu verwenden ist. Darüber hinaus werden viele Funktionen wie beispielsweise Sinus, Wurzel und Logarithmus mitgeliefert.

JSON in Java

Um die JSON Dateien lesen und beschreiben zu können, verwendet der Parser die Bibliothek *JSON in Java*. Die Bibliothek parsed den String der JSON Datei in eine eigene Datenstruktur, welches anschließend vom programmeigenen Parser in ein Datenmodell geparst wird.

4.2 Klassendiagramm für das AutomatonSystem

Das in Abbildung 4.1 zu sehende Klassendiagramm zeigt den Aufbau des Datenmodells. Dabei kann das Klassendiagramm im gesamten als ein Hybrides System gesehen werden. Das Klassendiagramm beschränkt sich auf das Wesentliche, sodass diverse interne Variablen von dieser Darstellung weggelassen wurden.

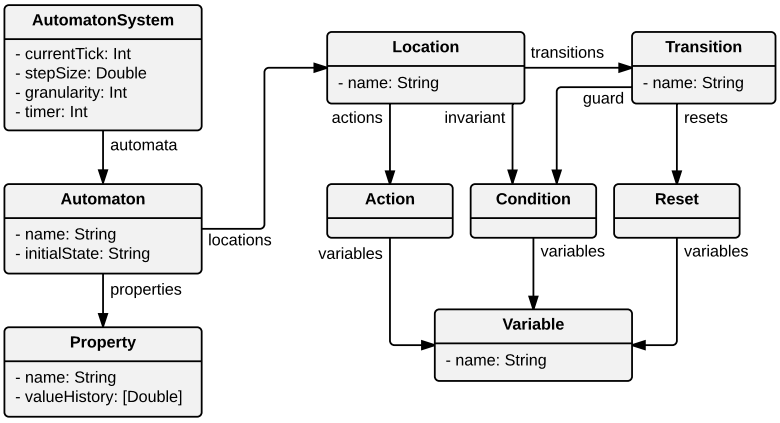

Abbildung 4.1 Klassendiagramm AutomatonSystem

Die Kernklasse ist *AutomatonSystem*, welches diverse Variablen für die Simulation beinhaltet und alle Automaten über die Beziehung *automata* hält. Über AutomatonSystem kann zu allen andern Klassen und deren Eigenschaften gelangt werden, sodass das Halten einer Referenz zum AutomatonSystem im Programm genügt. Die Klasse Automaton beinhaltet neben dem Initialzustand alle zum Automaten zugeordneten Properties. Dabei hält jede Property eine eigene Liste mit allen über den Simulationsverlauf errechneten Werte. Dies beinhaltet ebenfalls den aktuellen Wert. Automaton besitzt darüber hinaus eine Beziehung zu allen zugehörigen Locations. Diese wiederum besitzen Actions, welche die Activities repräsentieren und somit die Differentialgleichungen. Diese Actions werden in jedem Simulationsschritt ausgewertet. Des Weiteren beinhaltet eine Location eine Condition, welche im Wesentlichen einen Booleschen Ausdruck evaluiert. Im Falle der Location wird die Condition als Invariante verwendet. Des Weiteren existiert die Klasse Transition. Location hat auch zu dieser Klasse hier eine Beziehung, welche alle Transitionen einer Location verwaltet. Eine Transition besitzt einen Guard, welcher ebenfalls von der Klasse Condition repräsentiert wird. Außerdem hat die Klasse Transition Resets, welche bei der Verwendung einer Transition ausgeführt werden und die Werte der Properties eines Automaten verändern. Abschließend existiert noch die Helfer-Klasse Variable. Diese dient dazu, die in einer Action, Condition oder Reset stehenden Variablennamen ausfindig zu machen und dadurch die entsprechende Evaluierung der korrekten Property zuordnen zu können.

4.3 Simulationsprozess

Die Simulation eines Laufs kann in Simon sowohl manuell als auch automatisch durchgeführt werden. Im manuellen Modus wählt der Nutzer die Transition bzw. den Verbleib in der aktuellen Location nach jedem Tick selbst aus. Im automatischen Betrieb wählt das Tool mit einer gleichverteilten Wahrscheinlichkeit über alle Möglichkeiten zufällig selbst den nächsten Schritt. Technisch besteht allerdings zwischen beiden Varianten kein großer Unterschied. Im Falle der manuellen Simulation wird in der Klasse *AutomatonSystem* die Funktion *performTick()* aufgerufen, welche einen kompletten Simulationsschritt durchführt, wenn der *Next Tick*-Button geklickt wird. In der automatischen Simulation wird die Methode hingegen mithilfe eines Timers automatisch aufgerufen, welcher in dem Intervall ausgelöst wird, welches der Anwender im *Control Panel* unter *Simulation Speed* angegeben hat. Die Simulation besteht somit aus einer Verkettung von Simulationsschritten.

Ein einzelner Simulationsschritt wird in Abbildung 4.2 in einem Flussdiagramm gezeigt, welcher im Folgenden näher erläutert wird.

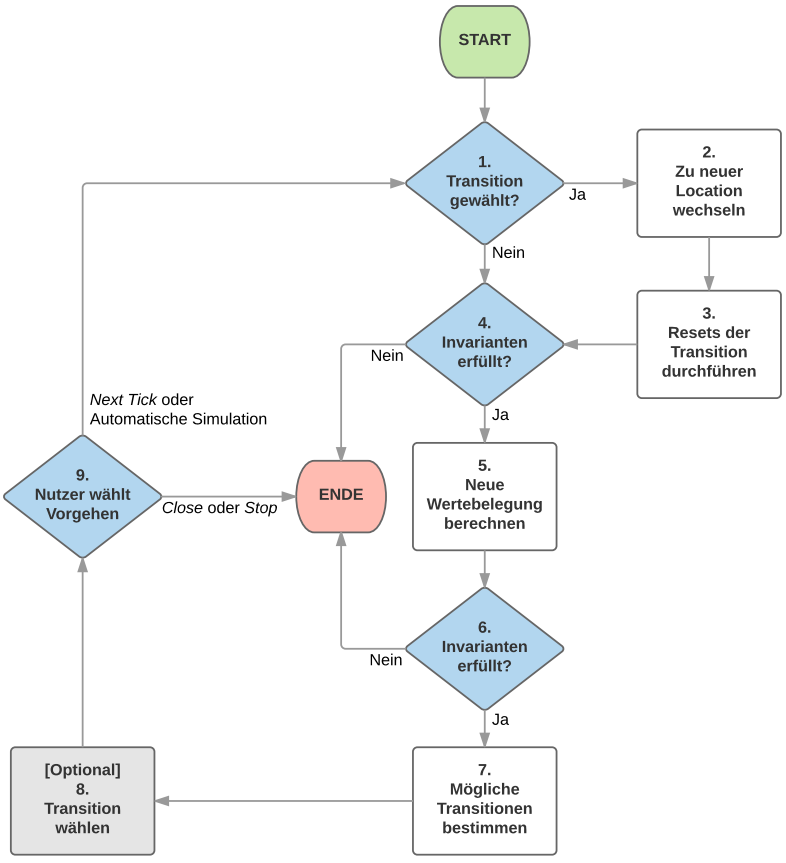

Abbildung 4.2 Simulationsschritt

1. Transition gewählt?

Zum Start der Simulation existiert keine „gewählte Transition“. Es werden stattdessen die vom Nutzer in der Eingabemaske definierten Initialwerte gesetzt und die initiale Location gewählt. Sollte sich die Simulation jedoch in einem späteren Zeitpunkt mit vorangegangenen Simulationsschritten befinden, könnte eine Transition gewählt worden sein.

2. Zu neuer Location wechseln

Liegt eine solche Transition vor, wird zu der neuen Location gewechselt. Die Invariante der neuen Location muss zu diesem Zeitpunkt noch nicht überprüft werden, da nur valide Transitionen zur Wahl standen, in denen die Invarianten der Ziellocation erfüllt sind. Somit ist die gewählte Transition an dieser Stelle sicher.

3. Resets der Transition durchführen

Nun werden die Resets der gewählten Transition durchgeführt und dadurch die Wertebelegung verändert.

4. Invarianten erfüllt?

An dieser Stelle werden die Invarianten zum ersten Mal überprüft. Sollte mindestens eine Invariante nicht erfüllt sein, wird die Simulation sofort beendet, da ein undefinierter Zustand vorliegt. Das Verhalten des Systems ist in diesem Fall nicht definiert und deshalb eine korrekte Simulation nicht mehr gewährleistet.

5. Neue Wertebelegung berechnen

In diesem Schritt werden die neuen Wertebelegungen der Variablen berechnet. Dabei werden diese für den Zeitpunkt $t = t_i + h$ mit t_i als Ausgangszeitpunkt und h als vom Anwender in der Simulationsansicht definierten Schrittweite berechnet. Die Berechnung wird in der vom Anwender gewählten Granularität durchgeführt und daher in entsprechend viele Teilberechnungen aufgeteilt, um der exakten Lösungskurve näher zu kommen. Die Berechnung selbst wird mithilfe von Näherungsverfahren durchgeführt, welche im Grundlagen Kapitel bereits beschrieben wurden. Dem Anwender stehen hierfür sowohl das Euler- als auch das klassische Runge-Kutta-Verfahren zur Verfügung.

Dies ist der einzige Schritt im gesamten Simulationsschritt, in dem tatsächlich Zeit vergeht. Alle anderen Schritte werden sofort ausgeführt und es vergeht dabei keine Zeit. Dies schließt den Schritt 2 ein, in der die Location gewechselt wird.

6. Invarianten erfüllt?

Hier findet eine erneute Überprüfung der Invarianten statt. Aufgrund der neuen Wertebelegung ist es möglich, dass die Invarianten nicht mehr erfüllt sind. Sollte dies der Fall sein, wird die Simulation auch hier sofort beendet.

7. Mögliche Transitionen bestimmen

Nun werden die möglichen Transitionen bestimmt. Es ist auch möglich, dass keine Transition zur Wahl steht. In diesem Fall kann das System jedoch in der aktuellen Location verweilen, da in Schritt 7 die Invariante bereits überprüft wurde.

8. Transition wählen (optional)

Optional kann der Anwender nun, sollten Transitionen zur Auswahl stehen, eine Transition wählen. Das System kann jedoch trotz einer möglichen Transition auch in der aktuellen Location verweilen.

9. Nutzer wählt Vorgehen

Hat der Anwender die automatische Simulation aktiviert, wird die Simulation automatisch fortgesetzt und der nächste Simulationsschritt startet in Schritt 1. Alternativ kann der Anwender durch das Klicken des *Next Tick*-Buttons den nächsten Simulationsschritt manuell einleiten. Klickt der Anwender hingegen auf den *Stop*-Button, wird die Simulation beendet. Das gleiche passiert, wenn der Anwender auf den *Close*-Button tippt, um die Simulation zu beenden und die Simulationsansicht zu schließen.

5 Simon - Das Simulationstool

Das Simulationstool *Simon* wurde mit der plattformübergreifenden Programmiersprache *Java* geschrieben und unter Verwendung des Frameworks *JavaFX* um eine grafische Benutzeroberfläche ergänzt. Das Programm besteht im Wesentlichen aus zwei Teilen: Der *Eingabemaske* sowie der *Simulationsansicht*.

5.1 Eingabemaske

In der Eingabemaske können Anwender von Simon die Hybriden Systeme eingeben und mit Initialwerten belegen. Die eingegebenen Systeme können zudem gespeichert und wieder geöffnet werden. So müssen die Daten zum Einen bei der Verwendung nicht immer wieder neu eingegeben werden. Zum Anderen können auf diese Weise große Hybride Systeme über einen längeren Zeitraum hinweg partiell eingegeben werden. Zudem können Duplikate von bereits vorhandenen Systemen angelegt werden und als Basis für neue Eingaben dienen. Wenn also beispielsweise mehrere, leicht unterschiedliche Systeme simuliert werden sollen, können diese mithilfe von Kopien der Originaldatei schneller in das Tool eingegeben werden. Die Daten werden im JSON Format abgespeichert und anhand der syntaktischen Struktur aus [Yör17] aufgebaut. Die Eingabemaske hat zwei Hauptreiter: *Definition* und *Configuration*.

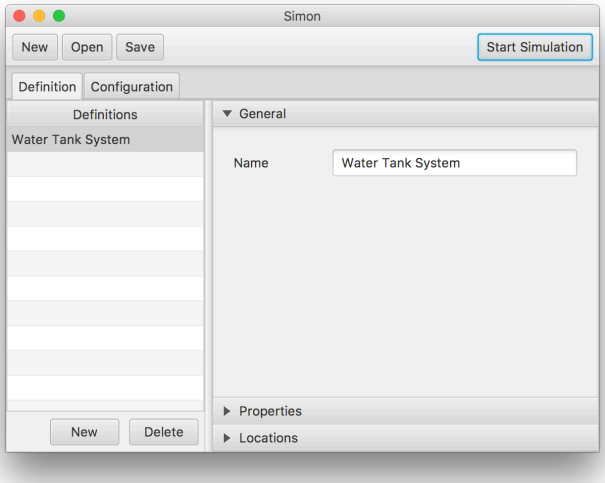

Abbildung 5.1 Screenshot: Definition Reiter

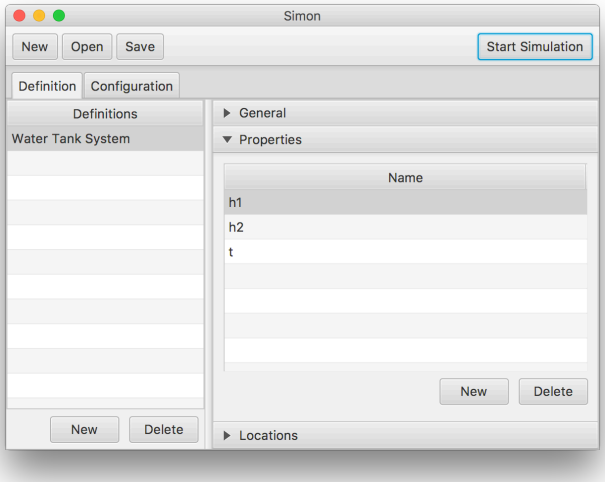

Abbildung 5.2 Screenshot: Properties

Im Definition-Reiter werden die Hybriden Systeme als Vorlagen definiert. Diese Vorlagen werden in Simon *Definitionen* genannt werden. Abbildungen 5.1 bis 5.3 zeigen den Inhalt des Definition-Reiter. Die Listenansicht links zeigt die Auflistung der Vorlagen. Klickt der Anwender auf eine Vorlage, wird ihm auf der rechten Seite die Detailansicht des gewählten Hybriden Systems angezeigt, in der der Anwender die entsprechende Vorlage einsehen und bearbeiten kann. Die Ansicht ist dabei in einer sogenannten Akkordion-Ansicht organisiert, welche als eine horizontale Variante der Reiter-Ansicht gesehen werden kann. Diese besteht aus den ausklappbaren Teilen *General*, *Properties* und *Locations*. Unter *General* lassen sich allgemeine Punkte editieren. Derzeit beinhaltet das lediglich den Namen des Moduls. Allgemeine Punkte, wie z.B. eine Farbe zur Unterscheidung der verschiedenen Module auf der Simulationsansicht, könnten eine mögliche Ergänzung zu einem späteren Zeitpunkt sein. Unter *Properties*, zu sehen in Abbildung 5.2, werden die (globalen) Variablen des Moduls angelegt und aufgelistet. Als Datentyp stehen derzeit lediglich Gleitkommazahlen zur Verfügung, welche in dem

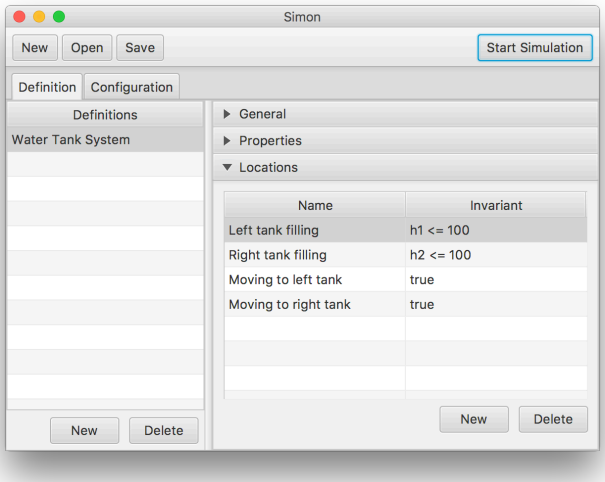

Abbildung 5.3 Screenshot: Locations

Tool intern als *double* verwaltet werden. Unter dem dritten Element der Detailansicht namens *Locations* lassen sich die diskreten Zustände der Module verwalten (siehe Abbildung 5.3). So eine Location besteht dabei aus einem Namen, einer Invariante, einer Activity und den Transitionen. Die Eingabemaske hierzu kann Abbildung 5.4 entnommen werden.

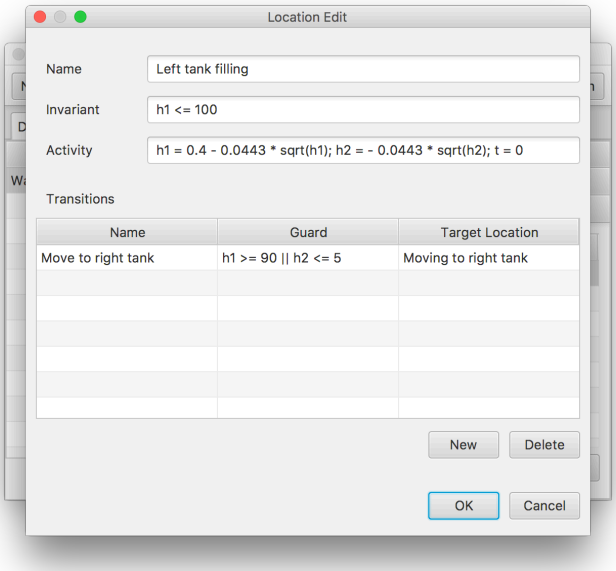

Abbildung 5.4 Screenshot: Location Eingabemaske

Die Invarianten werden als Boolesche Ausdrücke akzeptiert und können mit *AND*- und *OR*-Operatoren (umgesetzt mit *&&* und ||) beliebig verkettet werden. Dabei gilt folgende Grammatik für die Invariante Inv:

- $Inv := x \circ y \mid Inv \,\&\&\, Inv \mid Inv \,||\, Inv$
- mit $x, y \in \mathbb{R}$ und $\circ \in \{ <, >, <=, >=, == \}$

Die Activity der Location wird als eine Liste von gewöhnlichen Differentialgleichungen erster Ordnung angegeben. Für jede Variable wird hierbei eine Gleichung benötigt. Dabei werden die einzelnen Differentialgleichungen Semikolon-separiert in das entsprechende Textfeld in expliziter Form eingetragen. Es stehen zudem diverse Funktionen bereit, die mit der Programmbibliothek *exp4j* mitgeliefert werden, welche unter [Ass17] zu finden sind. So können beispielsweise die Wurzel- und Sinusfunktion verwendet werden. Damit die Eingabe von Simon verarbeitet werden kann, sind diverse Einschränkungen und Regeln zu beachten:

- Es muss für jede Variable genau eine explizite Differentialgleichung erster Ordnung angegeben werden.
- Das Zeichen der Ableitung wird nicht angegeben.
- Zahlen können aus $\mathbb{R}$ eingegeben werden. Dabei wird die englische Schreibweise von Dezimalzahlen angewandt, sodass ein Punkt als Dezimal-Trennzeichen verwendet wird.

- Es sind die gängigen Operationen mit +, −, * und / erlaubt. Zudem sind die erwähnten Funktionen unter [Ass17] anwendbar.
- Ebenso sind Verschachtelungen mit (und) erlaubt.

Sollen beispielsweise die Differentialgleichungen

$$y'(t) = 2{,}15 \cdot \sqrt{y(t)} \text{ und } z'(t) = \big(z(t) - 0{,}5\big) * 3$$

eingegeben werden, wird folgende Eingabe benötigt:

$$y = 2.15 * sqrt(y)\,;\;\; z = (z - 0.5) * 3$$

Die Transitionen werden in einer eigenen Tabellen-Ansicht organisiert (siehe Abbildung 5.4) und beinhalten in einer eigenen Detail-Ansicht, die in Form eines Popups bei Bedarf erscheint (siehe Abbildung 5.5), einen Namen, einen Guard, eine Target Location und den Reset. Wie bei den Invarianten der Locations auch, lassen sich die Guards mit *AND*- und *OR*-Operatoren verketten. Auch können Semikolon-separiert mehrere Resets angegeben werden.

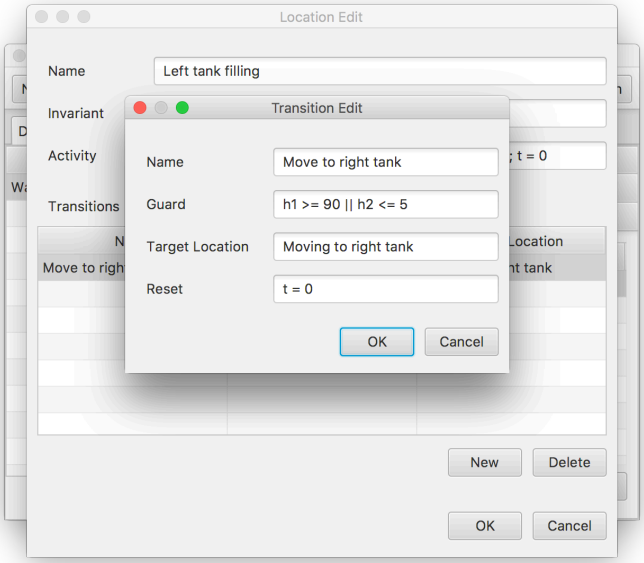

Abbildung 5.5 Screenshot: Transition Eingabemaske

Der Configuration-Reiter (siehe Abbildung 5.6), welcher wie der Definition-Reiter mit einer Listenansicht links und einer Detailansicht rechts organisiert wird, dient zur Erzeugung konkreter Instanzen. Dabei handelt es sich um Instanzen der zuvor definierten Module. Betrachten ließen sich die einzelnen Instanzen als eigenständige Hybride Systeme. In der Detailansicht zu jeder Instanz werden der Name der Modul-Definition, der Name der Instanz selbst sowie der Initialzustand (bzw. die Initial-Location) und die initiale Wertebelegung der Properties beschrieben.

Die Kombination aus der Eingabe der Definition und Konfiguration ergibt dann das Hybride System, welches anschließend zur Simulation genutzt werden kann. Ist das Hybride System vollständig definiert, lässt sich über den *Start Simulation*-Button oben

rechts in der Eingabemaske die Simulationsansicht starten, welche im Folgenden näher beschrieben wird.

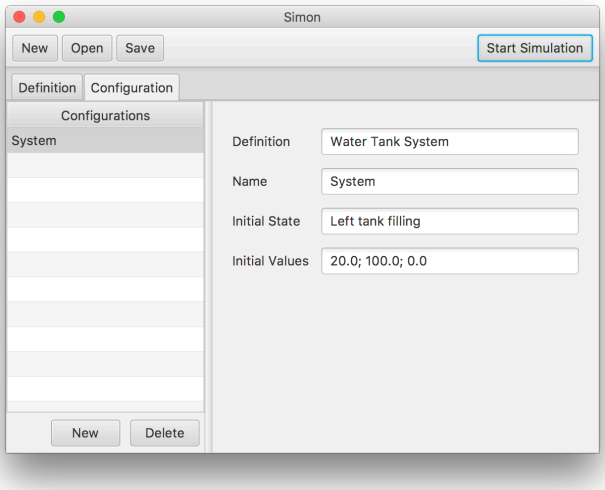

Abbildung 5.6 Screenshot: Configuration Reiter

5.2 Simulationsansicht

In der Simulationsansicht, welche in Abbildung 5.7 zu sehen ist, findet die eigentliche Simulation des spezifischen Hybriden Systems statt. Dem Anwender stehen hierfür diverse Bereiche zur Verfügung, die in diesem Abschnitt näher beschrieben werden.

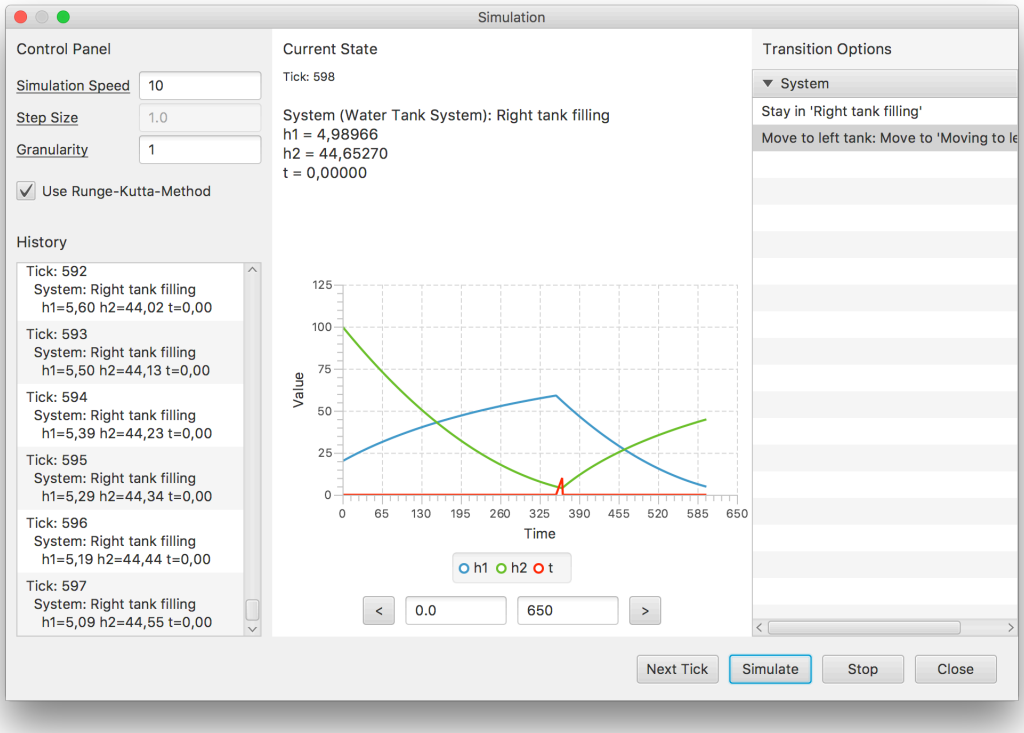

Abbildung 5.7 Screenshot: Simulationsansicht

Im *Control Panel*-Bereich oben links können die Parameter der Simulation definiert werden.

Die *Simulation-Speed* gibt an, in welchem Abstand die Ticks auf der Benutzeroberfläche angezeigt werden sollen. Die Angabe ist in Millisekunden. Eine kleinere Zahl bedeutet also, dass die Ticks in einem kürzeren Abstand berechnet und angezeigt werden. Diese Angabe beeinflusst das Ergebnis bzw. den Verlauf der Simulation in keiner Weise. Es führt nur dazu, dass die Simulation schneller oder langsamer berechnet wird.

Die *Step-Size* beschreibt die Schrittweite jedes einzelnen Ticks. Die Ausgabe erfolgt also in den in der Step-Size angegebenen Schritten. Eine kleinere Schrittweite bedeutet an dieser Stelle, dass in kleineren Zwischenschritten ausgegeben wird und die Simulation feiner und somit genauer wird. Eine zu große Schrittweite kann zudem dazu führen, dass das System unnötigerweise in einen undefinierten Zustand gelangt, bevor es eine Transition nehmen konnte. Dies passiert, wenn zu einem Zeitpunkt die Invariante der aktuellen Location nicht erfüllt ist. Mit einer kleineren Schrittweite wäre es unter Umständen möglich gewesen, zuvor eine Transition zu wählen, welche den Verlauf der Simulation beeinflusst hätte.

Der Wert hinter *Granularity* gibt an, wie viele Zwischenschritte in jedem einzelnen Tick berechnet werden sollen. Eine höhere Granularity bedeutet also eine feinere und genauere Simulation bei gleichzeitig erhöhtem Rechenaufwand. So kann der Anwender bestimmen, wie fein die Lösungskurve nachgezeichnet werden soll. Daraus leitet sich ab, dass sich die eigentliche Schrittweite der Simulation aus der Kombination der Angaben unter Step-Size und Granularity mit $h = \frac{step\ size}{granularity}$ ergibt. Für die Berechnung selbst könnte also auf die Angabe der Granularity verzichtet und die Schrittweite h direkt angegeben werden. Dies würde jedoch den Benutzer einschränken. Denn abhängig von den Anforderungen an die Simulation oder das Hybride System, kann zwar eine genaue Simulation mit einer sehr kleinen Schrittweite gewünscht sein. Jedoch würde ohne die separate Angabe von Schrittweite und Granularität die Ausgabe auch entsprechend häufiger erfolgen. Mit der Aufteilung der Schrittweite hat der Anwender also auf Wunsch eine feinere Möglichkeit, den Simulationsverlauf und das Ausgabeintervall zu steuern. Im Control Panel besteht darüber hinaus noch die Möglichkeit, zwischen dem Euler-Verfahren und dem klassischen Runge-Kutta-Verfahren zu wählen. Standardmäßig ist das Runge-Kutta-Verfahren ausgewählt. Wählt der Anwender diese Option ab, wird stattdessen das Euler-Verfahren angewandt. Das gewählte Verfahren wird zur Berechnung der Activities verwendet, wie in Abschnitt 4.3 beschrieben.

In der Simulationsansicht befindet sich auf der linken Seite unter der Control-Panel die *History*-Listenansicht, welcher den vergangenen Verlauf der Simulation anzeigt. Dem Anwender werden in der History chronologisch der jeweilige Tick zusammen mit der Location und der Wertebelegung der Variablen aufgelistet. Wenn eine Transition stattfand, werden beide Locations mit einem Pfeil getrennt nebeneinander dargestellt.

Im unteren Bereich befinden sich bis zu vier Aktions-Buttons zur Steuerung der Simulation. Der *Stop*-Button stoppt die Simulation und setzt alle Werte zum Startzeitpunkt zurück. Dies beinhaltet den Tick, die Wertebelegungen der Variablen und die gewählte Location. Der *Close*-Button stoppt die Simulation und schließt zudem die

Simulationsansicht, sodass der Anwender zurück in die Eingabemaske gelangt. Der *Simulate*-Button startet die automatische Simulation. Dabei werden die Transitions zufällig gewählt und die Simulation solange fortgesetzt, bis die Invariante der gewählten Location nicht mehr erfüllt ist und somit nicht mehr korrekt weiter simuliert werden kann. Die Berechnung wird in dem vom Anwender unter Simulation Speed im Control Panel gewählten Interval durchgeführt. Drückt der Anwender auf den Simulate-Button, wird dieser zu einem *Pause*-Button, mit dem der Anwender die Simulation zu jedem Zeitpunkt anhalten kann. Wenn der Anwender die Simulation pausiert, wird entsprechend der Pause-Button wieder zum Simulate-Button, mit dem der Anwender die automatische Simulation wieder fortsetzen kann. Zuletzt gibt es noch den *Next Tick*-Button. Dieser erlaubt die Simulation manuell zu durchlaufen. Die manuelle Simulation steht nur im pausierten Zustand zur Verfügung und wird versteckt, wenn der Anwender den Simulate-Button geklickt hat.

In der manuellen Simulation ist insbesondere der rechte Teil der Simulations-Ansicht von Interesse: Der *Transition Options*-Bereich. Hier sieht der Anwender in jedem Tick alle möglichen Transitionen und die vom System zufällig gewählte Transition. Als Option besteht zudem die Möglichkeit, in der aktuellen Location zu verweilen und somit keine Transition zu wählen. Ein Eingreifen des Anwenders ist hier in der manuellen Variante der Simulation möglich. Wird die Simulation automatisch durchgeführt und ein entsprechend niedriger Simulation Speed gewählt, kann der Anwender auch in der automatischen Variante eingreifen.

Im *Current State*-Bereich wird der aktuelle Zustand des Systems angezeigt. Hierzu gehört die Angabe des Ticks und die Auflistung aller Systeme mit deren aktuellen Locations und Wertebelegungen. Direkt unter diesem Bereich wird ein Liniendiagramm angezeigt, welches den Verlauf der Variablenwerte grafisch darstellt. Unter dem Liniendiagramm kann der Anwender den anzuzeigenden zeitlichen Ausschnitt definieren, indem er die Anfangs- und Endzeit angibt. Mit den Pfeil-Buttons rechts und links neben den Textfeldern besteht zudem die Möglichkeit, zwischen den Ausschnitten zu blättern.

6 Ergebnis

6.1 Erreichte Ziele

Im Folgenden werden die in der Einleitung definierten Ziele kritisch mit dem entstandenen Produkt verglichen und diskutiert.

Simulations-Tool für naturwissenschaftliche Experimente

Das Ziel dieser Masterarbeit war die Entwicklung eines Tools zur Simulation von naturwissenschaftlichen Experimenten. Dieses Ziel konnte mit Simon erreicht werden. Dabei wurde jedoch die Beschränkung auf naturwissenschaftliche Experimente vermieden, indem im allgemeinen Hybride Systeme als Simulationsmodell zum Einsatz kommen, welche ihre Evolution mit gewöhnlichen Differentialgleichungen erster Ordnung beschreiben. Dies deckt einen großen Teil der naturwissenschaftlichen Experimente ab und ermöglicht Simulationen auch unabhängig davon. Jedoch können durch den Ausschluss von Differentialgleichungen höherer Ordnung gewisse Experimente nicht simuliert werden, wie z.B. solche mit Beschleunigungsvorgängen.

Das Tool sollte gemäß den Zielen von Lehrern und Schülern bedienbar sein. Die Lehrer benötigen zur Verwendung von Simon jedoch Vorwissen über Differentialgleichungen und Hybride Systeme. Darüber hinaus ist die umgesetzte grafische Benutzeroberfläche aus Kapazitätsgründen simpel und könnte benutzerfreundlicher gestaltet sein, indem beispielsweise dem Nutzer Fehleingaben direkt angezeigt werden. Des Weiteren kommen häufig Textfelder zum Einsatz. An Stellen, an denen eine vordefinierte Auswahl existiert, eignen sich beispielsweise Dropdown-Listen besser, um Fehleingaben im Vorfeld zu vermeiden. Ansonsten lässt sich Simon jedoch potenziell von Lehrern und Schülern anwenden.

Abkapselung von der iPad App

In Kapitel 1 wurde in den Zielen genannt, dass das Tool abgekapselt von der iPad App entwickelt werden soll, um den Einsatzzweck nicht auf die App zu beschränken. Dieses Ziel wurde erreicht, indem Simon mit der plattformunabhängigen Programmiersprache *Java* geschrieben wurde. Das Tool kann damit betriebssystemunabhängig verwendet werden. Eine Portierung zu einer Smartphone bzw. Tablet App ist jedoch ohne Anpassungen an der Benutzeroberfläche nicht möglich, da JavaFX als Framework für die grafische Benutzeroberfläche zum Einsatz kam, welches auf Desktop-Betriebssysteme wie *Apple macOS*, *Microsoft Windows* oder diverse *Linux Desktop-Umgebungen* ausgelegt ist.

Einbau diverser Entscheidungslogiken

Ein weiteres Ziel war der Einbau diverser Entscheidungslogiken zur Behandlung von Nichtdeterminismus. Um den Nichtdeterminismus abfangen zu können, wurde zunächst eine Zeitdiskretisierung eingeführt. Diese hat den in Kapitel 3 beschriebenen kontinuierlichen Nichtdeterminismus aufgelöst. Um den diskreten Nichtdeterminismus aufzulösen, wurden die in den Zielen definierten Entscheidungslogiken eingebaut. Demnach kann der Nutzer zwischen einer automatischen Simulation - indem vom Tool ein zufälliger Pfad gewählt wird - und einer manuellen Simulation wählen. In der manuellen Simulation entscheidet der Nutzer nach jedem Tick, ob und welche (valide) Transition gewählt werden soll.

Es wurde zusätzlich in den Zielen erwähnt, dass neben diesen beiden Logiken weitere hinzugefügt werden können, wenn das Tool dahingehend in der Zukunft erweitert wird. Dies wurde in der Entwicklung von Simon berücksichtigt, sodass prinzipiell eine Erweiterung möglich ist. Da Simon jedoch zusätzlich über eine grafische Benutzeroberfläche verfügt, muss die erweiterte Entscheidungslogik auch hier eingebettet werden, um sie dem Nutzer zugänglich zu machen. Je nach implementierter Erweiterung kann dies auch umfangreiche Anpassungen nach sich ziehen.

6.2 Ausblick

Simon ermöglicht die gewünschte Simulation von Hybriden Systemen. Dabei wurde während der Bearbeitung der Masterarbeit wert auf ein korrekt funktionierendes Tool gelegt und dabei in Kauf genommen, dass der Funktionsumfang geringer ist, wenn im Gegenzug das Produkt möglichst fehlerfrei und funktionsfähig ist. Während der Entwicklung sind mehrere Ideen für Erweiterungen und Verbesserungen von Simon entstanden, die aus Zeit- und Kapazitätsgründen nicht umgesetzt werden konnten. Auf diese wird im Folgenden näher eingegangen.

Allgemeine Verbesserungen des Tools

Wie bereits in Abschnitt 6.1 erwähnt, besteht die Ansicht zur Eingabe der Hybriden Systeme und der Initialwerte im Wesentlichen aus Textfeldern. Diese sind für freie Texte wie die Bezeichnung von Variablen oder Transitionsnamen gut geeignet. Wenn allerdings beispielsweise die Startlocation ausgewählt werden soll, wäre eine Listenansicht oder eine Dropdown-Liste besser geeignet. Dem Nutzer wird so die genaue Schreibweise abgenommen und er kann die Eingabe schneller tätigen. Dadurch werden Fehler in der Eingabe vermieden.

Ebenso gibt es Verbesserungsmöglichkeiten bei der Eingabe von Booleschen Ausdrücken, wie sie bei den Invarianten der Locations und bei den Guards der Transitions vorkommen. Hier ist eine direkte Evaluierung der Eingabe denkbar, um dem Nutzer unmittelbar zu signalisieren, dass die Eingabe nicht korrekt ist. Auch wäre eine automatische Vervollständigung von eingegebenen Variablennamen denkbar, ähnlich wie bei einer Entwicklungsumgebung für Programmiersprachen. Dies wäre auch anwendbar für die Eingabe der Differentialgleichungen in der Activity der Location.

Verbesserungsmöglichkeiten dieser Art lassen sich unter dem Sammelbegriff *Benutzerfreundlichkeit* zusammenfassen. Solche lassen sich an einigen weiteren Stellen in Simon finden. Benutzerfreundlichkeit ist wichtig, gerade wenn Anwender ohne größeres technisches Hintergrundwissen eine Software verwenden sollen. Jedoch wurde auf das Thema bei der Umsetzung dieses Tools bewusst nicht intensiv eingegangen. Benutzerfreundlichkeit-steigernde Maßnahmen können sehr zeitaufwendig werden, was unter Umständen Einbußen im Funktionsumfang nach sich gezogen hätte. Mit einer grafischen Benutzeroberfläche (Simon wäre auch als Kommandozeilen-Anwendung ohne grafische Benutzeroberfläche realisierbar gewesen) wurde bereits ein großer Schritt für eine bessere Bedienbarkeit getan. Des Weiteren wurden viele weitere Verbesserungen eingebaut, die sich als nützlich und als nicht allzu zeitaufwendig herausgestellt haben. Das Liniendiagramm in der Simulationsansicht ist so ein Beispiel. Der Zeitaufwand hielt sich in Grenzen, hingegen wurde mit dem Einbau einer solchen Ansicht das Tool aufgewertet.

Verbesserte Eingabe großer Hybrider Systeme

Die Eingabe von kleinen Hybriden Systemen mit wenigen Variablen ist mit der realisierten Eingabemaske problemlos möglich. Es existieren jedoch auch Hybride Systeme mit mehreren hundert Variablen. Die Eingabe hierfür wäre sehr zeitaufwendig. Um dem entgegenzuwirken, wäre eine mögliche Verbesserung von Simon eine bessere Eingabemöglichkeit größerer Systeme, indem Variablenlisten eingegeben werden können, die eine bestimmte Variable mit mehreren Instanzen beschreiben. Beispielsweise könnte ein System zur Simulation des Verhaltens einer Verkehrskreuzung viele Fahrzeuge benötigen. Die Idee zur verbesserten Eingabe wäre an dieser Stelle, mit Indizes zu arbeiten, um nicht für jedes Fahrzeug eine eigene Variable *fahrzeug0*, *fahrzeug1*, *fahrzeug2*,… anlegen zu müssen. Im Tool wäre der Zugriff auf ein bestimmtes Fahrzeug über die Notation *fahrzeug[k]* denkbar.

Halbautomatische Simulation

Die manuelle Simulation ist momentan so realisiert, dass der Anwender die Ticks nacheinander mit einem Klick auf einen Button durchlaufen kann. Dabei hat der Nutzer auch die Möglichkeit, eine Transition zu wählen und so die zufällige Auswahl des Tools zu verändern. Das Problem an der manuellen Simulation ist jedoch, dass je nach gewählter Schrittweite viele Ticks vergehen können, ehe eine Wahlmöglichkeit auftaucht. Es kann passieren, dass das System auf diese Weise lange in einer bestimmten Location verweilt, ohne eine valide Transition zum Wechsel der Location angeboten zu bekommen.

Hier wäre eine *halbautomatische Simulation* denkbar, in der solange automatisch simuliert wird, bis eine Wahlmöglichkeit auftaucht. So kann der Nutzer in den Ablauf der Simulation eingreifen, ohne jeden einzelnen Tick durchlaufen zu müssen.

Schrittweitensteuerung

Die gewählte Schrittweite bestimmt im Wesentlichen die Genauigkeit der Simulation. Eine kleinere Schrittweite verbessert die approximierte Lösung, bei gleichzeitig er-

höhtem Rechenaufwand. Eine sinnvolle Erweiterung hierzu wäre eine *variable* Schrittweite mit Hilfe einer Schrittweitensteuerung. Eine Schrittweitensteuerung analysiert die Lösungskurve und berechnet so die Schrittweite für den nächsten Schritt voraus. Dabei werden in der Regel für kleine Unterschiede in den Werten zwischen den Schritten große Schrittweiten und für große Unterschiede kleine Schrittweiten gewählt, um eine möglichst genaue Approximation mit möglichst wenig Schritten zu realisieren.

Folgende Simulation wäre beispielsweise mit einer Schrittweitensteuerung effizienter: Betrachtet wird ein System, welches automatisch Pflanzen bewässert. Die Bewässerung der Pflanzen dauert wenige Sekunden. Die Phase der Nicht-Bewässerung kann hingegen Tage dauern. Wenn für die Simulation eines solchen Systems eine konstante Schrittweite zum Einsatz kommen soll, muss diese so gewählt werden, dass die (kürzere) Bewässerungs-Phase in mehreren Schritten erfasst wird. Denkbar wäre beispielsweise eine Schrittweite von einer Sekunde. Dies würde jedoch dazu führen, dass die Schrittweite in der Phase der Nicht-Bewässerung in einem kleinen Sekundentakt simuliert wird, in der die Unterschiede in den Werten zwischen den Schritten sehr gering bis kaum messbar sind. Eine sinnvolle variable Schrittweite wäre in einem solchen System beispielsweise 0,1 Sekunden für die Bewässerung und 30 Minuten für die Nicht-Bewässerung.

Benutzeroberflächen für verschiedene Szenarien

Eine weitere Möglichkeit zur Erweiterung von Simon wären spezifische Benutzeroberflächen für verschiedene Anwendungsszenarien. Um das in der Motivation erwähnte Thema der naturwissenschaftlichen Experimente aufzugreifen: Denkbar wäre eine Benutzeroberfläche, speziell für bestimme Experimente. Derzeit werden jegliche Hybride Systeme als Eingabe akzeptiert, was das Tool zwar mächtiger macht, jedoch nicht einfacher in der Verwendung. Wird die Eingabemaske auf die Eingabe von Experimenten begrenzt, können beispielsweise dem Anwender Elemente wie *Temperatur*, *Höhe* und *Durchmesser* zur Hand gegeben werden, womit Variablen näher beschrieben werden können. Simon würde daraufhin wissen, dass es sich bei einer bestimmten Variable um die Temperatur handelt und kann die Ausgabe dementsprechend anders darstellen, beispielsweise mit einem Thermometer, um aus der Simulationsansicht die Abstraktion abzuschwächen und so für Schüler verständlicher zu machen.

Es wäre ebenso möglich, die Benutzeroberfläche noch weiter einzuschränken, sodass das Tool die Simulation eines *bestimmten* naturwissenschaftlichen Experiments erlaubt. Mit dieser Einschränkung könnte die Benutzeroberfläche sehr simpel gestaltet werden, indem beispielsweise die für das Experiment benötigten Elemente als Grafiken auf der Benutzeroberfläche liegen. So könnten Messbecher, Thermometer etc. als Grafiken auf einem virtuellen Tisch liegen und über Interaktionen von den Schülern verwendet werden. Dies würde zwar das Tool stark einschränken, jedoch die Anwendung für bestimmte Einsatzzwecke erheblich verbessern und vereinfachen.

Differentialgleichungen

Simon unterstützt aus Komplexitätsgründen lediglich explizite gewöhnliche Differentialgleichungen erster Ordnung. Dabei kann in jeder Location für jede Variable genau eine Differentialgleichung angegeben werden. In Kapitel 2 wurde erwähnt, dass sich jede gewöhnliche Differentialgleichung *n*-ter Ordnung in ein System von gewöhnlichen Differentialgleichungen erster Ordnung überführen lässt. Deshalb wäre es interessant, das Tool um die Unterstützung von Systemen gewöhnlicher Differentialgleichungen erster Ordnung zu erweitern.

Interessant wäre auch die Erweiterung von Simon zur nativen Unterstützung von Differentialgleichungen *n*-ter Ordnung. Dies ruft allerdings neue Fragestellungen hervor, welche gelöst werden müssten, ehe Simon dahingehend erweitert werden könnte. Auch wäre eine Erweiterung auf partielle Differentialgleichungen denkbar.

Eine weitere sinnvolle Funktion wäre das exakte Lösen für bestimmte Klassen von Differentialgleichungen. Das spart zum Einen Rechenzeit und liefert zum Anderen exakte Ergebnisse und keine Approximationen. Dies kann mit den bestehenden numerischen Lösungsverfahren kombiniert werden, sodass diese weiterhin verwendet werden, wenn eine exakte Lösung nicht berechnet werden kann.

Parallelbetrieb

Bereits in Simon vorbereitet, aber aufgrund des Zeitaufwands nicht final umgesetzt, ist der Parallelbetrieb von Hybriden System. Es können bereits mehrere Hybride Systeme eingegeben und nebeneinander simuliert werden. Auch ist programmintern der Zugriff auf Variablenwerte zwischen den Systemen vorbereitet und von Anfang an vorgesehen worden. Jedoch konnte der Beweis, dass solche parallelen Systeme korrekt arbeiten, aus Zeitgründen nicht weiter verfolgt werden. Das Thema erwies sich als zu komplex. Jedoch wäre eine Erweiterung in diese Richtung durchaus interessant und sinnvoll.

6.3 Fazit

Das Thema der Simulation von Hybriden System ist umfangreich und interessant. In dieser Masterarbeit konnte ein Tool entwickelt werden, welches Hybride Systeme simulieren kann. Simon hat zwar einen kleinen Funktionsumfang, kann aber bereits in diesem Zustand eingesetzt werden. Die meisten Ziele wurden dabei erreicht und konnten während der Umsetzung um ein paar weitere interessante Aspekte, wie das Liniendiagramm oder die History in der Simulationsansicht, erweitert werden.

Beachtenswert ist jedoch auch, dass während der Entwicklung viele Ideen für mögliche Erweiterungen von Simon entstanden sind, welche im vorangegangenen Abschnitt 6.2 beschrieben wurden. Das Thema scheint damit noch sehr unerschöpft zu sein und bietet viel Potenzial für weitere interessante Arbeiten.

Literaturverzeichnis

[YY16] Lara Yörük und Orcun Yörük. Projekt im Fachgebiet Theoretische Informatik/Formale Methoden in Zusammenarbeit mit dem Fachgebiet Didaktik der Biologie: *Apple iPad Applikation DiVoX,* Universität Kassel, 2016

[MM11] Monique Meier und Jürgen Mayer. *Gewusst Vee! - Ein Diagnoseinstrument zur Erfassung von Konzept- und Methodenwissen im Biologieunterricht*, 2011

[ABB01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, u.a. *UPPAAL - Now, Next, and Future*, 2001

[UPP15] UPPAAL. *Published Material*, unter: http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml, zuletzt geändert am 10.05.2015

[BLL97] Johan Bengtsson, Kim Larsen, Fredrik Larsson, u.a. *New Generation of UPPAAL*, 1997

[Gun10] Prof. Roland Gunesch. *Einführung in Dynamische Systeme*, Universität Hamburg, 2010

[Kaw17] Christoph Kawan. *Vorlesungsskript Dynamische Systeme*, Universität Passau, 2017

[Bos13] Hartmut Bossel. *Simulation dynamischer Systeme: Grundwissen, Methoden, Programme*, Springer-Verlag, 2013

[GJ09] Lars Grüne und Oliver Junge. *Gewöhnliche Differentialgleichungen,* Vieweg+Teubner, 2009

[Col81] L. Collatz. *Differentialgleichungen*, Teubner, 1981

[But16] J. C. Butcher. *Numerical Methods for Ordinary Differential Equations,* Wiley, 2016

[Neu13] Werner Neundorf. *Numerik gewöhnlicher Differentialgleichungen mit Computeralgebrasystemen,* Universitätsverlag Ilmenau, 2013

[Met02] Dr. Wolfgang Metzler. *Numerische Mathematik II,* 2002

[Hen96] T. A. Henzinger. *The Theory of Hybrid Automata*, IEEE Computer Society, 1996

[Cro06] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON),* JSON.org, 2006

[Yör17] Lara Yörük. *SoPHY - A specification language for hybrid systems,* 2017

[GJS00] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification - Second Edition,* Addison Wesley, 2000

[Har96] Stephan Hartmann. *The World as a Process: Simulations in the Natural and Social Sciences*, 1996

[LTS08] John Lygeros, Claire Tomlin, and Shankar Sastry. *Hybrid Systems: Modeling, Analysis and Control*, 2008

[Ass17] Frank Asseg. *exp4j,* unter: http://objecthunter.net/exp4j/, zuletzt geändert am 30.01.2017

[And94] M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Lund Institute of Technology, 1994