CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Graphische Benutzeroberfläche für HyKeY

Henning Rohlfs

Carl von Ossietzky Universität Oldenburg
Fakultät II
Department für Informatik
Abteilung Entwicklung korrekter Systeme

Inhaltsverzeichnis

1 Einleitung

KeYmaera ist ein Programm für die computerunterstützte Verifikation von hybriden Systemen. Bis jetzt war die einzige Möglichkeit hybride Systeme in einem für KeYmaera verständlichen Format einzugeben die Benutzung eines Texteditors. Da die manuelle Eingabe aber sowohl langsam als auch fehleranfällig ist, ist das Ziel dieses Projekts die Entwicklung eines Werkzeugs mit welchem die Eingabe intuitiv und schnell möglich ist. Dies wird erreicht durch die Implementierung einer grafischen Benutzeroberfläche. Um zusätzlich die Wiederverwertbarkeit von den so erzeugten hybriden Systemen zu erhöhen, wird ein Export und Import in das PHAVer Format geschaffen.

Dieses Kapitel fasst das Grundwissen zusammen, welches benötigt wird, um das Projekt zu verstehen.

Ein *hybrides System* ist ein mathematisches Modell, das sowohl kontinuierliche wie auch diskrete Eigenschaften aufweist [1]. Ein *hybrider Automat* [1] ist eine Möglichkeit, ein hybrides System oder Teile davon darzustellen. Die Kanten des Automaten werden dabei genutzt, um das diskrete Verhalten des Systems abzubilden, während die Knoten das kontinuierliche Verhalten abbilden.

JFlex ist ein Generator für lexikalische Scanner (kurz Lexer). Der erzeugte Scanner ist eine Java Klasse, welche eine Eingabe in Form von Text in eine Abfolge von so genannten Token zerlegt. Ein Token ist ein Grundbestandteil der zu übersetztenden Sprache, beispielsweise ein Operator, ein Schlüsselwort oder eine Variable.

Mit Hilfe von *JavaCup* lässt sich ein Parser generieren. Ein Parser ist derjenige Teil eines Übersetzers, welcher aus einer Eingabe ein zur Weiterverarbeitung nützliches Format erzeugt. Die Eingabe besteht dabei aus Token, welche von einem lexikalischen Scanner erzeugt werden. Dazu erzeugt JavaCup aus einer Spezifikation einen LALR Parser. Der Parser wird in Form einer Java Klasse erstellt, was es ermöglicht, ihn einfach innerhalb eines Java Programms zu benutzen.

HySat-GUI ist ein Programm, welches als Projekt an der Carl von Ossietzky Universität Oldenburg entwickelt wurde. Es ermöglicht die grafische Eingabe von hybriden Automaten, welche dann in die Sprachen HLang und iSat übersetzt werden können. Mehrere Automaten können bei HySat-GUI als Teil eines hybriden Systems eingegeben werden.

HySat-GUI besteht aus mehreren Paketen, die nach ihrer Funktionalität benannt sind. Die *Paketstruktur* ist in Tabelle 1 beschrieben.

Das von HySat-GUI verwendete *Datenmodell* ist hierarchisch aufgebaut. Das hybride System ist dargestellt durch `HybridSystem`. Ein System enthält hybride Automaten in Form von `HybridAutomaton`. Jeder Automat wiederum besteht aus Kanten und Knoten. Ein Knoten wird dabei durch `State` abgebildet, eine Kante durch `Transition`.

Paket	Enthält ...
data	das Modell sowie den Export.
data.hybrid_expression*	das beim Export benutzte Zwischenmodell für Formeln.
help	für die Hilfe benötigte Klassen.
java_cup, org.jdom	Abhängigkeiten.
parser	die mit JavaCup und JFlex generierten Parser und Lexer.
ui*	die klasse, welche zur grafischen Darstellung notwendig sind.
xml	die zum Speichern und Laden benötigten Klassen.

* gilt auch für Unterpakete

Tabelle 1: Paketstruktur von HySat-GUI

KeYmaera und *PHAVer* sind Werkzeuge zur Verifikation von hybriden Programmen. KeYmaera ist eine Verbindung von KeY[2] und Mathematica.

Zum besseren Verständnis der nachfolgenden Kapitel folgt nun zunächst ein kurzer *Entwurfsüberblick*.
Um die Ziele des Projekts zu erreichen, wird HySat-GUI erweitert werden. Zunächst wird, um die Benutzbarkeit des Programms zu erhöhen, eine Undo Funktion implementiert (siehe 3.1). Damit bezeichnet man die Möglichkeit, ausgeführte Aktionen rückgängig zu machen beziehungsweise zu wiederholen. Hierfür wird eine Umstrukturierung von HySat-GUI notwendig sein.
Die Kompatibilität mit KeYmaera wird durch einen Export in das von KeYmaera genutzte Dateiformat erreicht. Da HySat-GUI bereits verschiedene Exporte unterstützt, bietet es sich diese Erweiterung die nötige Grundstruktur, wodurch sich gröÃere Änderungen am bestehenden Quellcode vermeiden lassen.
Weiterhin wird ein Export und Import für PHAVer Dateien implementiert. Dies wird die Wiederverwendbarkeit der erzeugten hybriden Systeme stark erhöhen. Der Export wird ähnlich wie der KeYmaera Export gröÃtenteils unabhängig vom restlichen Code realisiert. Für die Implementierung des Imports wird ein mit JavaCup generierter Parser zusammen mit einem mit JFlex generierten lexikalischen Scanner genutzt.

2 Syntax und Semantik

2.1 Hybride Automaten in HySat-GUI

HySat-GUI erlaubt das Modellieren eines hybriden Systems als Parallelprodukt hybrider Automaten. Das verwendete Modell wird in diesem Kapitel nach [3] beschrieben.

Ein *hybrides System* in HySat-GUI besteht aus einer Deklaration (siehe 2.1), ein oder mehreren Automaten und einem Target.

Die *Systemdeklaration* besteht aus der Deklaration des Systems und seiner Variablen. In der Systemdeklaration wird spezifiziert, aus welchen Automaten das System zusammengesetzt ist. Die Syntax hierfür ist `system=<automatonlist>;`, wobei `<automatonlist>` eine durch Kommata getrennte Aufzählung von Automaten ist. Ein hybrides System ist also ein Tupel S von Automaten sowie globalen Variablen: $S = (A_1, ..., A_n, V), n \in \mathbb{N}$ wobei A_1 bis A_n Automaten sind und V eine Menge von Variablen, die in allen Automaten genutzt werden können.

Die *Deklaration einer Variablen* geschieht in der Syntax `[const] <typ> <name>;`. HySat-GUI kennt die Typen `int`, `real` und `bool`. Durch die Typen wird der Wertebereich einer Variablen festgelegt. Variablen des Typs `int` haben den Wertebereich $\mathbb{N}$, `real` Variablen sind aus $\mathbb{R}$. Variablen vom Typ `bool` sind aus der Menge $true, false$. Bei der Übersetzung von Systemvariablen in das Zwischenformat wird dem Namen jeder Variablen der Präfix `system_variable_` vorangestellt, um bei der anschließenden Übersetzung in das Zielformat eine eindeutige Identifizierung der Variablen zu ermöglichen.

Ein *hybrider Automat* ist durch seine Zustände, Transitionen und seine Deklaration spezifiziert. In HySat-GUI wird er als einen Graphen bestehend aus Knoten und Kanten dargestellt. Ein Automat kann entweder in einem Zustand verweilen oder einer Transition in einen anderen Zustand folgen.
Formal ist ein hybrider Automat A ein Tupel $A = (\Sigma, T, R, inv, l, u, m, g, a, init)$, mit

- Σ Zustandsmenge
- T Menge der Transitionen
- R Menge der Variablen
- inv Menge an Bedingungen, Invariantbedingungen $inv = (inv_\sigma)_{\sigma \in \Sigma}$. Jeder Zustand kann eine Invariante besitzen. Ein Automat kann nur so lange in einem Zustand verweilen, wie die Invariante erfüllt ist.
- l untere Grenze der Steigung $l = (l_{\sigma,r})_{\sigma \in \Sigma, r \in R}$, die in jedem Zustand jeder Variablen eine minimale Steigung zuordnet
- u untere Grenze der Steigung $u = (u_{\sigma,r})_{\sigma \in \Sigma, r \in R}$, die in jedem Zustand jeder Variablen eine maximale Steigung zuordnet

- m eine totale Abbildung, welche jeder Transition einen Start- sowie Endzustand zuordnet. $m : T \to \Sigma^2$.
- g weist jeder Transition t einen Guard g mit $g = (g_t)_{t \in T}$ zu
- a weist jeder Transition t ein Update (oder `assignment`) a mit $a = (a_t)_{t \in T}$ zu.
- *init* eine Menge an Bedingungen *init* pro Zustand, welche bestimmen, ob der Automat in diesem Zustand starten kann: $init = (init_\sigma)_{\sigma \in \Sigma}$

Die Entwicklung eines Automaten kann entweder kontinuierlich als Fluss oder diskret als Sprung erfolgen.

Unter dem *Fluss* eines Automaten versteht man die kontinuierliche Entwicklung des Automaten. Der Fluss lässt sich durch das Tupel $(\sigma, \delta, \delta, x')$ darstellen. Dabei ist $\sigma \in \Sigma$ der Zustand, in dem sich der Automat befindet, $x \in (R^{\xrightarrow{total}}\mathbb{R})$ die die Startbelegung der Variablen, $x' \in (R^{\xrightarrow{total}}\mathbb{R})$ die Endbelegung der Zustandsvariablen und $\delta \in \mathbb{R}, \delta \geq 0$ die Verweildauer im Zustand σ. Dabei muss für die Start- sowie Endbelegung der Variablen die Invariante inv_σ erfüllt sein. Zudem muss die Steigung jeder Variablen r innerhalb der Grenzen $l_{\sigma,r}$ und $u_{\sigma,r}$ liegen.

Ein *Sprung* eines Automaten ist im Gegensatz zum Fluss eine diskrete Entwicklung. Ein Sprung kann als Tupel (σ, x, σ', o') ausgedrückt werden. $\sigma \in \Sigma$ ist dabei der Startzustand, $\sigma' \in \Sigma$ der Endzustand. Entsprechend ist $x \in (R^{\xrightarrow{total}}\mathbb{R})$ die Startbelegung der Variablen vorm Sprung und $x' \in (R^{\xrightarrow{total}}\mathbb{R})$ die Belegung der Variablen nach dem Sprung. Es muss dabei eine Transition t mit Abbildung $m(t) = (\sigma, \sigma')$ geben. Außerdem muss x den Guard $g(t)$ sowie x' das Update $a(t)$ der Transition t erfüllen.

In der *Deklaration eines Automaten* wird eine Liste von Variablen deklariert. Die Deklaration erfolgt mit der gleichen Syntax wie in der Systemdeklaration (siehe 2.1).

Das *Target* spezifiziert die Eigenschaft, die bei dem hybriden Systems nachgewiesen werden soll. In ihm kann die Menge an Endzuständen angegeben werden und es können Bedingungen an Variablen gestellt werden. Das Target wird in Form einer Formel bestehend aus boolschen Ausdrücken sowie mathematischen Vergleichen spezifiziert. Zustände sind dabei durch ihren Namen definiert und können wie eine boolsche Variable benutzt werden.

Ein *Zustand* ist ein Knoten eines hybriden Automaten. Während ein Automat in einem Zustand verweilt kann er eine kontinuierliche Änderung erfahren. Ein Zustand wird definiert durch seinen Namen, seine Startbedingung (auch Initialbedingung), seine Invariante, seine Flow-Bedingung sowie seine Dringlichkeit.

Die *Startbedingung* eines Zustandes ist diejenige Bedingung die erfüllt sein muss, damit ein Automat in diesem Zustand starten kann.

Eine *Invariante* ist eine Bedingung die an die Variablen eines Automaten gestellt wird. Ein Automat kann nur so lange in einem Zustand verweilen, wie dessen Invariante erfüllt ist.

Beim sogenannten *Flow* handelt es sich um ein Gleichungsssystem, welches angibt wie sich die Variablen eines Automaten entwickeln, während dieser in einem Zustand verweilt. Diese Entwicklung ist kontinuierlich.

Die *Dringlichkeit eines Zustandes* ist nur für KeYmaera von Bedeutung. Mit ihr wird angegeben, ob ein Automat gezwungen werden soll einen Zustand zu verlassen, sobald er dies kann. Muss ein Automat einer Transition aus einem Zustand so bald wie möglich folgen, so nennt man diesen Zustand `urgent`. Eine Markierung als `urgent` ist äquivalent dazu, alle Guards von allen Transitionen negiert zur Invariante des Zustands hinzuzufügen. Auch dadurch wird der Automat gezwungen den Zustand zu verlassen, sobald einer der Guards dies erlaubt.

Bei einer *Transition* handelt es sich um einen diskreten Sprung zwischen zwei Zuständen. In HySat-GUI wird sie durch eine Kante zwischen zwei Knoten dargestellt. Sie wird spezifiziert durch ihren Namen, einen Guard sowie ein Update.

Ein *Guard* ist eine Bedingung die erfüllt sein muss, damit ein Automat einer Transition folgen kann. Der Guard stellt Bedingungen an die Variablen des Automaten.
Über das *Update* (auch `assignment`) werden während des Übergangs zwischen zwei Zuständen die Variablen geändert. In HySat-GUI wird dies auch Assignment genannt.

2.2 Parallelkomposition von Automaten

In diesem Kapitel wird beschrieben, wie zwei Automaten durch Parallelkomposition zu einem einzelnen Automaten vereint werden können, ohne die Semantik des Systems zu ändern. Der beschriebene Algorithmus kann iterativ angewandt werden, um mehr als zwei Automaten zusammenzufassen. Dabei werden jeweils zwei Automaten miteinander verknüpft, so dass bei jedem Iterationsschritt die Anzahl der Automaten im System um eins sinkt. Wird dies oft genug wiederholt, so erhält man für ein hybrides System mit beliebig vielen Automaten ein semantisch gleichwertiges System, welches nur einen Automaten enthält.

Die Parallelkomposition von zwei Automaten geschieht in folgender Weise:

1. Finde einen neuen Namen
2. Erzeuge einen neuen Automaten
3. Erzeuge neue Zustände
4. Bestimme den neuen Startzustand
5. Erzeuge neue Transitionen

6. Bereinige Zustände, deren Ausgangszustände `urgent` sind.

7. Entferne die Automaten A und B aus dem System.

In der nachfolgenden Beschreibung der einzelnen Schritt werden die beiden Automaten mit Automat A beziehungsweise Automat B bezeichnet. Da die Semantik am Ende der Parallelkomposition der ursprünglichen Semantik entspricht, ist es egal, welcher Automat dabei als A und welcher als B bezeichnet wird. Der neu erzeugte Automat wird im Nachfolgenden als Automat M bezeichnet.

Der folgende Algorithmus kann zur *Namensgenerierung* für den neuen Automaten genutzt werden. Er garantiert, dass es zu keiner doppelten Namensvergabe (zum Beispiel bei Zustandsnamen) kommen kann.

1. Erzeuge einen Namen durch Konkatenation von Name von Automat A, Unterstrich und Name von Automat B.

2. Überprüfe für alle Automaten des Systems, ob es einen Automaten gibt, der mit dem erzeugten Namen beginnt.

   (a) Falls nein: Benutze den erzeugten Namen.

   (b) Falls ja: Füge eine beliebige Zahl an den Namen an. Wiederhole ab Schritt 2.

Anschließend wird ein neuer, leerer Automat erzeugt. Dieser wird zum gleichen hybriden System wie die Automaten A und B hinzugefügt. Der neue Automat erhält den zuvor erzeugten Namen.

Bei der Parallelkomposition das *Target* (siehe Abschnitt 2.3) nicht geändert wird, muss die Übersetzung angepasst werden. Wird im Target ein Zustand eines der Ursprungsautomaten referenziert, so müssen nach der Parallelkomposition stattdessen alle Zustände referenziert werden, welche bei der Erzeugung als Ursprung diesen Zutand hatten.
Dies wird mit Hilfe eines Beispiels verdeutlicht. Gegeben seien Automat A mit den Zuständen a1 und a2 sowie Automat B mit den Zuständen b1 und b2. Bei der Parallelkomposition werden dann die Zustände a1_b1, a1_b2, a2_b1 sowie a2_b2 erzeugt. Wird im Target der Zustand a1 referenziert, so muss nach der Parallelkomposition die Übersetzung stattdessen sowohl Zustand a1_b1 als auch Zustand a1_b2 referenzieren. In KeYmaera wird beispielsweise aus dem ursprünglichen (state=a1) das zusammengesetzte (state=a1_b1 | state=a1_b2).
Um diese Übersetzung zu ermöglichen, muss die Zuordnung zwischen dem erzeugten Zustand und den Ausgangszuständen gespeichert werden. Dies geschieht bei der Erzeugung der Zustände während der Parallelkomposition.

Für die *Erzeugung der Zustände* von Automat M kann dieser Algorithmus genutzt werden:

1. Iteriere über die Zustände in Automat A.
   (a) Iteriere über die Zustände in Automat B.
      i. Erzeuge einen neuen Zustand.
      ii. Erzeuge einen Namen für den neuen Zustand bestehend aus den beiden ursprünglichen Namen.
      iii. Füge zum neuen Automaten alle flow Prädikate beider Ausgangsautomaten hinzu.
      iv. Füge zum neuen Automaten alle Invarianten beider Ausgangsautomaten hinzu.
      v. Speichere die Zuordnung (Neuer Zustand, alte Zustände) für die Übersetzung des Targets.

Die *Transitionen* des neuen Automaten werden durch die Ausführung das hier beschriebenen Algorithmus erzeugt.

1. Iteriere über die Zustände in Automat A.
   (a) Iteriere über die Transitionen in Automat B.
      i. Erzeuge eine neue Transition
      ii. Erzeuge einen zusammengesetzten Namen aus dem Namen der Transition und des Zustandes.
      iii. Übernehme jump sowie update der Ausgangstransition.

Anschließend wird der Algorithmus für die Zustände aus Automat B und die Transitionen aus Automat A wiederholt.

Um die Semantik von Zuständen zu erhalten, die *urgent* sind, muss im Anschluss an die Erzeugung der Zustände und Transitionen die nachstehende Vorschrift befolgt werden.

1. Iteriere über alle Zustände des Automaten M.
   (a) Teste, ob der Ausgangszustand aus Automat A `urgent` ist.
   (b) Falls ja, iteriere über die Transitionen, die vom Ausgangszustand ausgehen.
      i. Füge den negierten Guard der Transition der Invariante des neuen Zustandes hinzu.

2.3 KeYmaera

Das KeYmaera Format wird anhand eines Beispiels in Listing 1 in Hinsicht auf die Übersetzung von einem hybriden Automaten aus HySat-GUI erläutert. Die nachfolgenden Referenzen auf Zeilennummern beziehen sich wenn nicht anders angegeben auf

\[a \] F	nach allen Ausführungen von a ist F wahr
\< a \> F	nach mindestens einer Ausführung von a ist F wahr
F -> G	Implikation: aus F folgt G
F <-> G	Äquivalenz: F ist äquivalent zu G
F | G	Disjunktion: F oder G
F & G	Konjunktion: F und G
!F	Negation: nicht F
pred	Ein arithmetischer Ausdruck

Tabelle 2: dℒ Formeln

dieses Listing. Grundsätzlich ist das Format aus den Grundblöcken \functions, \sorts, \settings sowie \problem zusammengesetzt. Der Inhalt jedes Blocks wird von geschweiften Klammern umschlossen.
Außerdem sind an jeder Stelle Kommentare erlaubt, welche von /* und */ umschlossen werden (siehe Zeilen 15 bis 21).
Da KeYmaera selbst die Parallelkomposition nicht kann, werden vor der Übersetzung in das KeYmaera Format alle Automaten zu einem verbunden. Dies geschieht durch die Parallelkomposition wie beschrieben im Abschnitt 2.2.

Im Block *functions* werden Funktionen deklariert, die nicht im Format weiter spezifiziert sind, aber im Problemblock genutzt werden können. Die Deklaration einer Funktion besteht aus dem Datentyp sowie ihrem Namen gefolgt einer von Klammern umgebenen und von Kommata getrennten Liste an Argumenten und zum Schluss von einem Semikolon (Zeile 6). Eine Funktion kann auch ohne Argumente deklariert werden (Zeile 7), in welchem Fall auch die Klammern weggelassen werden. Ein solcher Block kann in den Zeilen 5 bis 8 gesehen werden.

In dem optionalen Block *sorts* wird der Wertebereich spezifiziert. Momentan werden von KeYmaera nur reale Zahlen unterstützt. Im Beispiel ist dieser Block in den Zeilen 1 bis 3 zu finden.

Im ebenfalls optionalen Block *settings* werden programmspezifische Einstellungen gespeichert. Im Beispiel ist dieser Block in den Zeilen 10 bis 12 zu finden.

Im Problemblock (Zeilen 14 bis 70) wird das eigentliche hybride System spezifiziert. Er wird durch *problem* gekennzeichnet. Der Problemblock enthält eine einzelne dℒ Formel.
Für die Übersetzung sind insbesondere dℒ Formeln aus Tabelle 2 sowie die hybriden Programme aus Tabelle 3 notwendig, wobei F und G dℒ Formeln sind, a und b hybride Programme und x und y Variablen. Bei f handelt es sich um eine arithmetische Formel.

Nachfolgend wird die von KeYmaera erwartete dℒ *Formel* nach [4] beschrieben. dℒ steht für `differential logic`. Damit ist eine Erweiterung von dynamischer Logik gemeint, durch die eine kontinuierliche Entwicklung spezifiziert werden kann. Dies ist

a; b	Sequenzielle Komposition (a dann b)
a ++ b	Nichtdeterministische Auswahl zwischen a und b
a*	Nichtdeterministische Wiederholung von a (beliebige Anzahl von Wiederholungen)
?F	stellt den Anspruch, dass F wahr ist, bricht ansonsten ab
{x‘=t, y‘=s, f}	Differenzialgleichungssystem
x := *	nichtdeterministische Zuweisung eines beliebigen
x := t	Weist x den Wert t zu.
R x	Variablendeklaration der Variablen x.

Tabelle 3: hybride Programme

notwendig, um ein hybrides System zu modellieren. dℒ erlaubt sowohl diskrete Sprünge als auch kontinuierliche Entwicklung. Zudem können die beiden kombiniert werden. Die Semantik von Transitionen wird dabei durch Zuweisungen an eine `state` Variable erzielt. So eine Zuweisung kann beispielsweise in der Form $state := 5;$ erfolgen. Dadurch werden diskrete Sprünge zwischen den Zuständen eines hybriden Systems möglich. Die kontinuierliche Evolution des Systems wird durch Differenzialgleichungen spezifiziert. Eine solche Differenzialgleichung kann beispielsweise die Form $x' = 5;$ haben.

Ein in HySat-GUI hybrider Automat $A = (\Sigma, T, R, inv, l, u, m, g, a, init)$ lässt sich als dℒ Formel darstellen. Hierzu wird eine bijektive Zuordnung $f : \Sigma \to \mathbb{N}$ benötigt. Diese kann beliebig gewählt werden. Dadurch kann die Zustandsmenge Σ als Belegung einer Variablen *state* abgebildet werden. Zusätzlich zur Variablen *state* wird für jede Variabel $r, r \in R$ eine Variable in dℒ deklariert. Für jeden Zustand $\sigma, \sigma \in \Sigma$ wird eine nichtdeterministische Auswahl eingeführt. Durch eine Überprüfung der Variablen *state* wird sichergestellt, dass jeweils nur ein Zustand aktiv sein kann.

Der Fluss des Automaten wird als ein Zweig einer nichtdeterministischen Auswahl innerhalb eines so aufgebauten Zustandes dargestellt. Dazu werden in jedem Zustand σ die Invariantbedingungen inv_σ des Zustandes sowie die Steigung der Variablen (angegeben in HySat-GUI durch l bzw. u) in Form eines Differenzialgleichungssystems $\{x‘ = t, y‘ = s, f\}$ angegeben. f wird dabei durch die Invarianten inv_σ gebildet.

Die Sprünge des hybriden Automaten beim benutzen von Transitionen werden in Form der anderen Zweige der nichtdeterministischen Auswahl innerhalb eines Zustandes dargestellt. Für jede Transition $t, t \in T$ mit $m(t) = (\sigma_1, \sigma_2)$ wird ein Zweig der Auswahl im Zustand σ_1 erzeugt. Der Guard $g(t)$ wird als Anspruch in der Form $?g(t)$ geltend gemacht. Durch sequenzielle Komposition wird anschließend das Update $a(t)$ ausgeführt. Anschließend, ebenfalls durch sequenzielle Komposition, wird der Automat in den Endzustand σ_2 versetzt, indem die Zuweisung $state := f(\sigma_2)$ ausgeführt wird. Hierdurch können die Mengen T sowie die Zuordnungen m, g und a als Teil der dℒ Formel wiedergegeben werden.

Die Menge von Startbedingung *init* wird zur Übersetzung beschränkt. Es wird gefordert, dass *init* genau für einen Zustand σ_{init} $true$ ist und für alle anderen Zustände

$false$. Dann kann die Übersetzung als $state = f(\sigma_{init})$ erfolgen. Damit wird jeder Teil des Tupels, welches den Automaten ausmacht in der dℒ Formel wiedergegeben.

Der erste Schritt *Übersetzung* des Systems ist also die Zuordnung einer eindeutigen Identifikationsnummer zu jedem Zustand. Es wird folgendes Grundgerüst verwendet: gelten die Vorbedingungen (Startzustand, Variablendeklaration) dann folgt daraus, dass nach einem beziehungsweise jedem Durchlauf des hybriden Programms das Target gelten muss. Ob die Existenz einer Lösung oder die Richtigkeit jeder Lösung überprüft wird durch eine Einstellung des Benutzers kontrolliert. Das hybride Programm bildet den hybriden Automaten aus HySat-GUI ab.
Daraus ergibt sich folgende Grundlegende Syntax: `\[ decl \] (state = start) -> \[ automaton \] target` beziehungsweise `\[ decl \] (state = i) -> \< automaton \> target`.
In `decl` werden die Variablen deklariert. `start` ist der Wert, der dem Startzustand zugeordnet ist. Im hybriden Programm unter `automaton` wird das eigentliche System abgebildet. `target` ist das Target.

Zunächst werden die *Variablendeklarationen* übersetzt. Jede in HySat-GUI deklarierte Variable wird auch im KeYmaera Export deklariert (Zeile 23). Zusätzlich wird eine weitere Variable `state` deklariert, welche den Zustand markiert, in dem sich das System gerade befindet. Die Zuordnung der States zu dieser Variablen wird zur besseren Lesbarkeit in einem Kommentar wiedergegeben (Zeile 15). Die Variable `state` wird auf den Wert gesetzt, der den Anfangszustand darstellt.

Im Teil `automaton` wird eine nichtdeterministische Auswahl zwischen den Zuständen des *Automaten* getroffen. Jeder Zustand (Zeile 27) wird dadurch charakterisiert, dass die Variable `state` den ihm zugeordneten Wert haben muss. Dies wird dadurch sichergestellt, dass die Variable mit Hilfe des Konstrukts `?(state = N)` überprüft wird, wobei N die Nummer des Zustands ist. So wird auch sichergestellt, dass trotz nichtdeterministischer Auswahl immer nur ein Zustand aktiv sein kann.

Ein *Zustand* selbst wird wiederum durch eine nichtdeterministische Auswahl dargestellt (siehe Zeile 30). Die Alternativen für die Auswahl dabei sind die verschiedenen Transitionen, die vom Zustand ausgehen sowie die Möglichkeit, im Zustand zu verweilen (Zeile 31). Das Verweilen in einem Zustand ist durch ein Differenzialgleichungssystem gekennzeichnet, bei dem sich `state` nicht verändert.

Eine *Transition* hingegen wird abgebildet durch die sequenzielle Komposition vom Guard und anschließender Ausführung des Updates sowie Veränderung der `state`-Variablen: `?(guard); update; state=N`, wobei N die Nummer des Zielzustandes der Transition ist. Ein Beispiel hierfür findet sich in Zeile 29.

Das Update einer Transition in HySat-GUI ist ein Gleichungssystem mit *gestrichenen Variablen*. Um dieses Gleichungssystem in KeYmaera darzustellen, werden temporäre

Variablen benötigt. Für jede gestrichene Variable wird eine temporäre Variable deklariert. Jeder Variablen wird zunächst ein beliebiger Wert zugewiesen. Anschließend wird der Anspruch gestellt, dass nach Ersetzung der gestrichenen Variablen durch die temporären Variablen das ursprüngliche Update wahr ist. Im Anschluss daran wird den ungestrichenen Variablen jeweils der Wert der temporären Variablen zugewiesen.
Beispielsweise wird aus `x = 5 + x' + y'` durch Anwendung des beschriebenen Algorithmus `(R x2, y2; x2:=*; y2:=*; ?(x = 5 + x2 + y2); x:=x2; y:=y2)`. Dabei werden x' durch x2 und y' durch y2 nach Vorschrift ersetzt.
Für den Sonderfall, dass das Update in der Form `x' = z` ist, wobei z keine gestrichenen Variablen enthält, so gibt es eine Optimierung der Ausgabe. Bei diesem einfachen Fall kann eine einfache Zuweisung erfolgen. So wird aus `x' = 5 + x + y` anstatt `(R x2; x2:=*; ?(x2 = 5 + x + y); x:=x2)` die stark vereinfachte Form `(x := 5 + x + y)` erzeugt.

Das *Target* (Zeile 67) wird als eine $\mathtt{d}\mathcal{L}$ Formel dargestellt. Da HySat-GUI es erlaubt, Zustände wie boolsche Variablen zu verwenden, werden diese, falls sie vorkommen, übersetzt, indem die `state`-Variable mit der Identifikationsnummer des Zustandes verglichen wird.
So wird in dem Beispiel aus `waterlevel.On = true;` in HySat-GUI `(((state = 0))) <=> (true) ))` in KeYmaera. `waterlevel.On` wird dabei wie beschrieben durch `(state=0)` ersetzt. Zusätzlich wird, entsprechend der KeYmaera Syntax aus = beim der Biimplikation <=>.

Listing 1: KeYmaera Beispiel (Wassertank)

```
1 \sorts {
2   R;
3 }
4
5 \functions {
6   R min(R);
7   R f;
8 }
9
10 \settings {"
11
12 "}
13
14 \problem{
15 /*
16   0      - On
17   1      - SentOff
18   2      - SentOn
19   3      - Off
20   4      - Init
21 */
22
23 \[R waterlevel_variable_x; R waterlevel_variable_y; R state; state := 4\] ( (st
```

```
24 ->
25  \<
26   (
27     (?(state=0);
28       (
29            (?(( (waterlevel_variable_y) <=> (0.0) )); waterlevel_variable_x :=
   state := 1)
30         ++
31            ({waterlevel_variable_x‘ = 1.0, waterlevel_variable_y‘ = 1.0, waterl
32       )
33     )
34   ++
35     (?(state=1);
36       (
37            (?(( (waterlevel_variable_x) <=> (2.0) ));  state := 3)
38         ++
39            ({waterlevel_variable_x‘ = 1.0, waterlevel_variable_y‘ = 1.0, waterl
40       )
41     )
42   ++
43     (?(state=2);
44       (
45            (?(( (waterlevel_variable_x) <=> (2.0) ));  state := 0)
46         ++
47            ({waterlevel_variable_x‘ = 1.0, waterlevel_variable_y‘ = -2.0, water
48       )
49     )
50   ++
51     (?(state=3);
52       (
53            (?(( (waterlevel_variable_y) <=> (5.0) )); waterlevel_variable_x :=
   state := 2)
54         ++
55            ({waterlevel_variable_x‘ = 1.0, waterlevel_variable_y‘ = -2.0, water
56       )
57     )
58   ++
59     (?(state=4);
60       (
61            (?(true); waterlevel_variable_y := 1.0;  state := 0)
62         ++
63            ({true, true})
64       )
65     )
66    )*
67  \>(( (((state = 0))) <=> (true) ))
68 )
69
```

70 }

2.4 PHAVer

Das PHAVer Format ist im Aufbau dem internen Format von HySat-GUI sehr ähnlich. Es besteht aus Konstantendefinitionen, einem oder mehreren Automaten sowie weiteren Befehlen, die im Zuge dieses Projekts aber nicht bearbeitet werden. Die Erläuterung des Formats geschieht wieder anhand eines Beispiels. Die Zeilennummern in den nachfolgenden Ausführungen beziehen sich auf Listing 2. Die folgende formale Definition stammt aus [5].

Ein hybrider Automat in PHAVer kann als Tupel $A = (Loc, Var_S, Var_I, Var_O, Lab, \rightarrow, Act, Inv, Init)$ dargestellt werden.

- Loc Endliche Zustandsmenge
- Var_S Menge an Zustandsvariablen
- Var_I Menge an Eingabevariablen mit $Var_S \cap Var_I = \emptyset$
- Var_O Menge an Ausgabevariablen mit $Var_O \subseteq Var_S$
- Lab Menge der Synchronisationslabel
- $\rightarrow$ Die Menge der Transitionen: $\rightarrow \subseteq Loc \times Lab \times 2^{V(Var) \times V(Var)} \times Loc$. V(Var) bezeichnet dabei eine Menge von Zuordnungen $v : Var \rightarrow \mathbb{R}$. Var ist definiert als $Var = Var_S \cup Var_I$.
- Act Eine Zuordnung $Act : Loc \rightarrow 2^{act(Var)}$. Als act(Var) wird eine Menge von Zuordnungen $a : \mathbb{R}^{\geq 0} \rightarrow V(Var)$ bezeichnet.
- Inv Eine Zuordnung $Inv : Loc \rightarrow 2^{V(Var)}$
- $Init$ Initialzustände, $Init \subseteq Loc \times V(Var)$ mit $(l, v) \in Init \Rightarrow v \in Inv(l)$

Bei der Übersetzung aus dem hybriden Automaten von HySat-GUI gelten die folgenden Zuordnungen.

- Σ wird zu Loc
- Aus der Menge der Variablen R wird Var.
- Die Invarianten inv entsprechen Inv.
- Die kontinuierliche Evolution, die in HySat-GUI durch l und u begrenzt wird, wird in PHAVer durch Act spezifiziert.
- Nicht unterstützt von HySat-GUI werden die Synchronisationslabel Lab. Diese können nicht übersetzt werden. Daher werden sie auch bei der Übersetzung von Transitionen ignoriert.

- $\rightarrow$ in PHAVer ist eine Menge an Transitionen, die durch einen Startzustand, ein Synchronisationslabel, ein Guard, ein Update und einen Endzustand gekennzeichnet sind. Entsprechende Eigenschaften (außer dem Synchronisationslabel) besitzen auch die Transitionen in HySat-GUI. $\rightarrow$ wird in HySat-GUI also durch T, m, g und a ausgedrückt.
- *init* eine Menge an Bedingungen *init* pro Zustand, welche bestimmen, ob der Automat in diesem Zustand starten kann: $init = (init_\sigma)_{\sigma \in \Sigma}$

Durch diese Zuordnungen wird eine Übersetzung des HySat-GUI Modells nach PHAVer möglich. Die Syntax dafür stammt aus [6]. Im Nachfolgenden wird ein Algorithmus beschrieben, welcher für jeden Bestandteil von PHAVer die Übersetzung entsprechend der erläuterten Zuordnungen erlaubt.

Eine *Konstante* wird definiert durch ihren Name gefolgt von :=, dem Wert und einem Semikolon (siehe 1). Konstanten werden im HySat-GUI in die Systemdeklaration aufgenommen.

Die Spezifikation eines *Automaten* im PHAVer Format wird umschlossen von dem Schlüsselwort `automaton` gefolgt vom Namen des Automaten am Anfang (Zeile 3) und `end` am Ende (Zeile 13). Innerhalb des Automaten werden Kontrollvariablen, Eingabevariablen, Parameter sowie Synclabel deklariert (Zeilen 4 bis 7). Die Syntax dafür ist jeweils folgendermaßen: Auf den Deklarationstyp folgt ein Doppelpunkt, eine von Kommata getrennte Liste von Bezeichnern und ein Semikolon.
Anschließend wird eine Liste von Zuständen sowie die Startbedingung deklariert.

Ein *Zustand* in PHAVer besteht aus einer Zeile, die den Zustand selbst definiert (Zeile 9) sowie einer beliebigen Anzahl an Trabnsitionen, die vom Zustand ausgehen (Zeile 10).
Die Syntax für den Zustand ist `loc <name>: while <invariant> wait {<flow>}`. Dabei ist <name> der Name des Zustands, <invariant> die Invariante und <flow> Das Gleichungssystem, welches die kontinuierliche Entwicklung des Zustands definiert.

Eine *Transition* in PHAVer hat die Syntax `when <guard> sync <label> do <flow> goto <targetstate>;`. `<guard>` ist dabei der Guard der Transition, `<label>` ist das Synclabel, welches aber von HySat-GUI nicht unterstützt wird. `<flow>` ist eine Formel die das Update angibt, welches bei der Transition die Variablen spezifiziert. `<targetstate>` ist der Name des Zielzustands.

Eine *disjunktive Normalform* (kurz DNF) bezeichnet eine besondere Darstellung eine boolschen Formel. Eine Formel ist dann in der disjunktiven Normalform, wenn es sich um eine Disjunktion von Konjunktionstermen handelt. Ein Konjunktionsterm ist ein Term, der ausschließlich aus konjunktiven Verknüpfungen von Variablen (oder negierten Variablen) besteht. Eine Formel lässt sich in DNF überführen, indem zunächst eine Wahrheitstabelle über alle Variablen gebildet wird. Dazu wird jede mögliche Kombination von Werten für die Variablen getestet. Ergibt die Formel für eine Wertebelegung

Konjunktionsterm enthält ..	Verhalten
keinen Zustand	füge den Term der Initialbedingung aller Zustände hinzu
einen Zustand	füge den Term dem enthaltenen Zustand hinzu
zwei oder mehr Zustände	ignoriere den Term

Tabelle 4: Übersetzung von Konjunktionstermen

`true`, so wird die Variablenbelegung als Konjunktionsterm in die DNF aufgenommen. Ergibt die Formel `false` wird zur nächsten Belegung gesprungen.

Die *Startbedingung* des Automaten ist eine Formel, die von `initially:` und einem Semikolon umschlossen wird. Der Name eines Zustands kann dabei wie eine boolsche Variable verwendet werden.
Da im Gegensatz zu HySat-GUI die Initialbedingung auf Ebene des Automaten und nicht lokal für jeden Zustand definiert ist, muss die Startbedingung beim Import übersetzt werden. Hierzu wird sie zunächst in disjunktive Normalform gebracht. Die einzelnen Konjunktionsterme der Startbedingung werden dann wie in Tabelle 4 beschrieben behandelt.

Durch den Algorithmus zur Bildung der DNF kann es nicht vorkommen, dass ein Konjunktionsterm einen negierten Zustand enthält.

Beim Export entsprechend aus den Bedingungen jedes Zustands durch Verknüpfung mit dem Namen des Zustands ein Konjunktionsterm.

Listing 2: PHAVer Beispiel (bouncing ball)

```
1   g:=1;
2
3   automaton bouncing_ball
4           contr_var: x, v;
5           input_var: a, b;
6           parameter: c;
7           synclabs: jump;
8
9           loc state: while x>=0 & x<=10 & v<=10 & v>=-10 wait {x'==v & v'=
10                  when x==0 & v<0 sync jump do {v'==-v*0.5 & x'==x} goto
11
12          initially: state & x==2 & v==0;
13  end
```

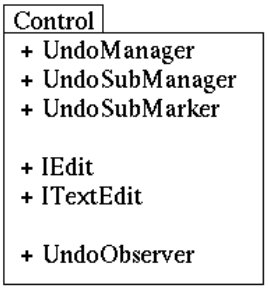

Abbildung 1: Wichtige Interfaces und Klassen im Paket control

3 Design und Implementierung

Da sich das Projekt in mehrere größtenteils unabhängige Unterprojekte aufteilen lässt, wird im Folgenden auch das Design und die Implementierung in diese Abschnitte aufgeteilt.

3.1 Undo und Redo

Dieser Teil des Projekts beschäftigt sich mit der Möglichkeit, ausgeführte Aktionen rückgängig zu machen und zu wiederholen. Da dies die größten Änderungen am bestehenden Programm mit sich bringt, wird diese Änderung zuerst entwickelt.

Beim *Entwurf* der Undo und Redo Funktionalität bietet sich das Command-Pattern [7] an. Hierbei werden alle Aktionen in sogenannte Commands gekapselt. Jedes Command kann sich selbst ausführen und die gemachten Änderungen rückgängig machen. Beispielsweise kann ein `AddTransitionCommand` eine Transition hinzufügen und selbige wieder löschen. Es speichert dazu alle nötigen Informationen wie den Start- und Endzustand sowie den Namen der Transition. Da das Command weiß, welche Transition es erzeugt hat, kann es diese auch wieder entfernen oder erneut hinzufügen.

Zusätzlich wird eine Liste der ausgeführten Aktionen bereitgehalten. Um den gesamten Zustand des Programms in einen früheren Zustand zu versetzen muss dann nur noch die Liste rückwärts durchgegangen werden und jedes Command dazu veranlasst werden, sich rückgängig zu machen.

Für die Commands sowie die beschriebene Liste wird ein neues Paket namens `control` erstellt. Die Liste wird, wie in Abschnitt 3.1.1 beschrieben, vom `UndoManager` in Verbindung mit dem `UndoSubManager` verwaltet. Alle Commands müssen das Interface `IEdit` implementieren. Für Änderungen an Texten steht das von `IEdit` abgeleitete Interface `ITextEdit` zur Verfügung.

Um einfach auf die Änderungen am Status des `UndoManagers` reagieren zu können, wird das Observer-Pattern [7] angewendet. Das Observer-Pattern ermöglicht es, Statusänderungen eines Objekts zu beobachten. Hierzu registrieren sich sogenannte Beobachter bei dem zu beobachteten Objekt. Ab diesem Zeitpunkt werden sie bei jeder Änderung des Status des Objekts benachrichtigt und können darauf reagieren. So

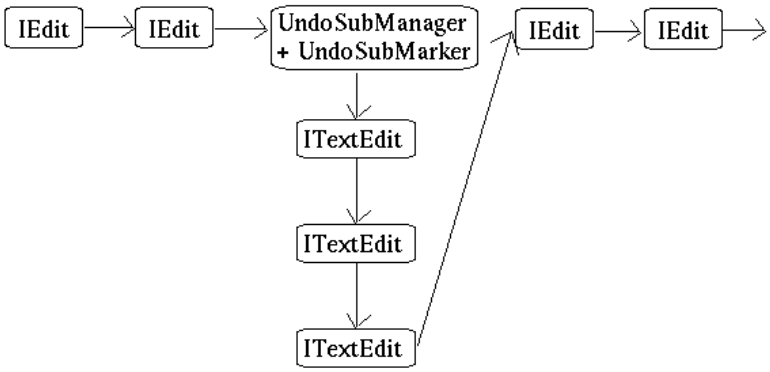

Abbildung 2: Aufbau der Liste im `UndoManager`

kann beispielsweise die Darstellung auf dem Bildschirm aktualisiert werden, wenn sich die dargestellten Daten verändern. So wird das Observer-Pattern auch bei HySat-GUI eingesetzt. Der `UndoManager` ist dabei das beobachtbare Objekt.

3.1.1 Implementierung

Bei der Implementierung wird der *`UndoManager`* dazu genutzt, eine Liste der Commands zu Verwalten. Er stellt die Funktionalität zum Hinzufügen, Ausführen und Rückgängigmachen von Commands zur Verfügung. Ebenso ermöglicht er es Beobachtern, sich als solche zu registrieren und wieder abzumelden.

Um den Speicherverbrauch auch nach längerer Benutzung zu begrenzen wird die Anzahl der im `UndoManager` gespeicherten Commands begrenzt. Wird diese Anzahl überschritten, so werden die ältesten Commands gelöscht. Da aber Änderungen an Texten schrittweise in Commands gespeichert werden, könnte es dazu kommen, dass nach längerem Editieren eines Textes alle anderen Änderungen verloren gehen und nur noch die Änderungen am Text rückgängig gemacht werden können. Um dies zu verhindern werden Textänderungen vom `UndoManager` folgendermaßen behandelt: Wird ein Command hinzugefügt, welches ITextEdit implementiert, so wird statt des Commands ein `UndoSubManager` sowie, aus Implementierungsgründen, ein `UndoSubMarker` in die Command-Liste eingefügt. Sollte bereits ein `UndoSubManager` als letztes in der Liste sein, so wird zunächst überprüft, ob dieser das neue Command akzeptiert und gegebenenfalls wird kein neuer `UndoSubManager` hinzugefügt. Das ursprüngliche Command wird dann an den `UndoSubManager` angehängt. Daraus ergibt sich eine Liste wie sie in Abbildung 2 dargestellt ist.

Dadurch wird erreicht, dass die Menge der Commands im `UndoManager` bei größeren Textänderungen nicht steigt und so auch danach noch sinnvolles Undo und Redo möglich ist.

`AbsoluteTextEdit` ist ein Command, welches mehrere Änderungen eines Textes speichern kann. Hierfür wird der Text vor sowie nach den Änderungen jeweils komplett gespeichert. Es erlaubt zusätzlich, weitere ITextEdits auf den gespeicherten Text anzuwenden und so weitere Änderungen hinzuzufügen.

Diese Klasse verwaltet ähnlich dem `UndoManager` eine Liste mit Commands. Im Unterschied zu diesem akzeptiert der `UndoSubManager` aber nur ITextEdits. Außerdem wird bei einem Überlauf der Liste das älteste Command nicht gelöscht. Stattdessen werden die beiden ältesten Commands zu einem Command verbunden, welches die Änderungen beider Commands enthält. Die Klasse, die hierfür genutzt wird, heißt `AbsoluteTextEdit`.

3.2 KeYmaera Export

Dieser Abschnitt beschreibt das *Design* des KeYmaera Exports. Da eines der Ziele beim Entwurf des Exports war, möglichst wenig Änderungen am ursprünglichen HySat-GUI zu verursachen, wurde das grundlegende Design von den bereits vorhandenen Exporten übernommen. Zunächst wird das vorhandene Hybride System in ein Zwischenmodell und anschließend in das Zielformat übersetzt. Das Zwischenmodell ist dabei für alle Exporte gleich.

Der erste Schritt ist die *Übersetzung in das Zwischenformat*. Das Zwischenmodell ist dem Modell von HySat-GUI sehr ähnlich. Es besteht ebenfalls aus einer das hybride System repräsentierenden Klasse (`ParseSystem`) welche die einzelnen Automaten enthält. Die Automaten werden durch die Klasse `HybridAutomaton` repräsentiert. Der `HybridAutomaton` enthält wiederum seine Zustände (`State`) und Transitionen (`Transition`). Bei der Übersetzung werden die im internen Modell noch als Text gespeicherten Formeln für das Zwischenmodell in einen hierarchischen Baum von Objekten konvertiert. Der Baum besteht aus Instanzen von Klassen, die das Interface `HybridExpression` implementieren. Es gibt entsprechende Klassen für alle benutzten Operatoren sowie für Variablen und Konstanten, beispielsweise `HybridExpressionPlus` und `HybridExpressionMinus`. Währenddessen wird außerdem in jedem `HybridAutomaton` eine Liste der deklarierten Variablen und Parameter erstellt.

Die Übersetzung erfolgt mit Hilfe von JavaCup und JFlex. Dafür werden verschiedene Parser und Lexer generiert, die jeweils für bestimmte Teile des hybriden Systems eingesetzt werden. So werden separate Parser für die Systemdeklaration, Parameter, Automatendeklaration, Variablen, Ausdrücke sowie das Target zur Verfügung.

Dieser erste Schritt ist größtenteils bereits von HySat-GUI implementiert und muss nur erweitert werden.

Da KeyMaera die *Parallelkomposition* von hybriden Programmen noch nicht beherrscht, werden beim Export in das KeyMaera-Format durch ein iteratives Verfahren alle Automaten des Systems vor der eigentlichen Übersetzung zu einem einzelnen Automaten zusammengefügt (siehe 2.2). Pro Iterationsschritt werden dabei zwei Automaten miteinander vereint. Dieser Schritt wird nach der Übersetzung in das Zwischenformat und vor der Übersetzung in das Zielformat durchgeführt. Diese Parallelkomposition ist beinahe transparent, das heißt für den Algorithmus zur Übersetzung gibt es nur wenige Unterschiede, ob ursprünglich ein oder mehrere Automaten im System vorhanden waren. Dadurch kann die Übersetzung soweit vereinfacht werden, dass für beide Fälle der gleiche Algorithmus verwendet wird.

Zur *Übersetzung in die Zielsprache* wurden bei HySat-GUI Felder in der Klasse `Statics` eingesetzt, welche durch ihren Wert signalisierten, welche Eigenschaften die Zielsprache haben sollte. Die Felder waren vom Typ `boolean`. Um den bestehenden Quellcode weiterhin benutzen zu können, wird dieses Konzept nur leicht angepasst. So wird statt eines `booleans` jetzt eine `enum` benutzt, welche die ursprünglichen Felder übernommen hat. Dadurch wurde die Leserlichkeit erleichtert, da nun eine neue Sprache als weiterer Eintrag im `enum` aufgelistet wird. Zudem ermöglicht dieser Ansatz es Code zu schreiben, welcher die Ausgabe in Abhängigkeit von der Zielsprache generieren kann.

3.2.1 Implementierung

Wie im Design beschrieben, steht am Anfang die *Übersetzung in das Zwischenformat*. Bei diesem Schritt wird wie beschrieben größtenteils der Code von HySat-GUI verwendet. Neu eingeführt werden zwei Operatoren: `HybridExpressionQuantExists` und `HybridExpressionQuantForall`. Diese repräsentieren die jeweiligen Quantoren. Um die Quantoren erkennen und übersetzen zu können, muss außerdem der ursprüngliche Parser für Ausdrücke spezialisiert werden.

In diesem Zuge wird auch das Makefile, eine Datei, die Spezifikationen für den Aufruf von `make` enthält, aktualisiert, mit welchem die Parser und Scanner generiert werden. Die Originalversion des Makefiles generierte bei jedem Aufruf alle Parser und Scanner neu. Da aber `make` gerade ein Programm ist, mit dem beim Bauen von Projekten nur geänderte Dateien neu kompiliert werden, wird das Makefile entsprechend angepasst und erzeugt nur dann neue Parser, wenn sich die jeweiligen Quelltexte ändern. Dies beschleunigt die eigentliche Implementierung der Änderungen am Parser und verkürzt so die Entwicklungszeit.

Um die Übersetzung von komponierten Automaten zu ermöglichen, wird für Variablen, die Zustände darstellen, eine neue Klasse eingeführt: `HybridBooleanStateVariable`. Diese wird im Target-Parser anstelle der vorher genutzten `NewHybridVariable` erzeugt. `HybridBooleanStateVariable` ist eine Spezialisierung von `NewHybridVariable`, was es ermöglicht, sie ohne weitere Änderungen am vorhandenen Code zu benutzen.

Die *Parallelkomposition von Automaten* wird in der Klasse `AbstractHybridProgramWriter` vorgenommen. Bei der Parallelkomposition wird zusätzlich zu dem Automaten eine Abbildung (in Form einer `Map`) erzeugt, welche die neu erzeugten Zustände auf die ursprünglichen Zustände abbildet. Anhand dieser Abbildung wird bei der Übersetzung in das KeyMaera-Format dazu benutzt, um die Endbedingung zu erzeugen. Wegen der Entscheidung, möglichst viel Code beizubehalten, geschieht dies in einer Methode, zu der keine Parameter übergeben werden können. Stattdessen wird mit Hilfe der `Map` in der Klasse `Statics` ein statisches Feld mit den nötigen Informationen gefüllt. Diese Informationen werden von den Instanzen der Klasse `HybridBooleanStateVariable` dazu genutzt, um die in 2.3 spezifizierte Übersetzung zu erzeugen.
Zusätzlich zu den im Algorithmus genannten Schritten muss dem neu erzeugten Automaten außerdem noch die Liste der Variablen beider Ausgangsautomaten hinzugefügt werden. Dies ist notwendig, da eine Variablenliste, welche bei der eigentlichen Übersetzung genutzt wird, bei der Erzeugung des Zwischenmodells generiert wird.

Im Anschluss an die Parallelkomposition findet die *Übersetzung in das Zielformat* statt. Hierzu wird in der Klasse `HybridProgramWriter` das Gerüst für das KeyMaera-Format geschrieben. Das heißt es werden die grundsätzlichen syntaktischen Konstrukte erzeugt. Diese werden mit den im Zwischenformat vorhandenen Formeln gefüllt. Dazu wird zunächst das Feld `Statics.language` auf die entsprechende Sprache gesetzt. Anschließend werden die `HybridExpressions` dazu veranlasst, ihre Textrepräsentation zu erzeugen. Da sie dies in Abhängigkeit von der gesetzten Sprache tun, erzeugt dies die benötigte Repräsentation der Formeln für das KeyMaera-Format.

Eine Ausnahme bildet das *Update*-Prädikat. Da KeyMaera gestrichene Variablen nicht auf der rechten Seite einer einfachen Zuweisung erlaubt, wird hier ein spezieller Algorithmus (siehe 2.3) zur Übersetzung genutzt, welcher sich allerdings nicht ohne weiteres mit dem bestehenden Code implementieren lässt. Daher wird dieser Fall in einer Methode in `HybridProgramWriter` behandelt. Da der Algorithmus aber eine Liste der gestrichenen Variablen benötigt, muss diese aus dem Zwischenmodell extrahiert werden. Hierzu werden die in den meistern Klassen, die `HybridExpression` implementieren so angepasst, dass sie rekursiv durchlaufen werden können. Um dies zu ermöglichen, werden die in den meisten Klassen bereits vorhandenen Methoden `get_right` und `get_left` verbindlich gemacht.

3.3 PHAVer Export

Der PHAVer Export ist in Hinsicht auf Design und Implementierung dem KeYmaera Export sehr ähnlich. In diesem Kapitel werden dementsprechend nur die Aspekte beschrieben, in denen sich die beiden Exporte unterscheiden.

Da PHAVer in der Lage ist mehrere Automaten in einer Datei zu behandeln wird auf die Parallelkomposition von Automaten verzichtet. Stattdessen werden die Automaten separat ausgegeben. Die Übersetzung erfolgt dabei entsprechend 2.4.
Zusätzlich zu den Übersetzungsschritten, die für den KeyMaera Export notwendig sind, müssen die Initialbedingungen übersetzt werden. Die Erzeugung der disjunktiven Normalform erfolgt dabei durch iterieren über die Zustände und disjunktive Verknüpfung der nach 2.4 erzeugten Formeln.

3.4 PHAVer Import

Wie beim *Entwurf* des KeYmaera Exports war es auch beim PHAVer Import das Ziel, möglichst große Mengen des bestehenden Quellcodes wiederzuverwenden. Aus diesem Grund besteht der Import aus einem Parser, welcher mit Hilfe von JavaCup und JFlex generiert wird. Die beim Parsen gewonnen Informationen werden in der Klasse `PhaverSystemBuilder` in das interne Datenformat von HyKey-GUI übertragen. Zusätzlich wird mit der Klasse `Phaver` eine einfache Schnittstelle zum Parser zur Verfügung gestellt.

Da sich das interne Datenmodell von HyKey-GUI nicht dazu eignet direkt mit Hilfe des Parsers generiert zu werden und da während der *Implementierung* große Änderungen am Code vermieden werden sollen, werden in der Klasse `PhaverSystemBuilder` die zusätzlich notwendigen Informationen während des Parsens gespeichert.
Zunächst werden in `Phaver` durch reguläre Ausdrücke aus der Eingabedatei die relevanten Informationen gewonnen. Dieser Ansatz ist nötig, da das PHAVer Format verschiedene Eigenschaften hat, die sich nicht in HyKey-GUI übertragen lassen. Durch Anwendung der regulären Ausdrücke lässt sich vermeiden, dass der Parser die unbehandelten Befehle als Eingabe erhält. Somit kann der Parser soweit vereinfacht werden, dass er nur die benutzten Daten erkennen muss.

Beim Import der *Initialbedingung* von Automaten kommt es zu den in 2.4 beschriebenen Problemen, da es bei HyKey-GUI eine Initialbedingung pro Zustand und bei PHAVer eine Initialbedingung pro Automat gibt. Wie beschrieben wird zunächst die disjunktive Normalform aus der eingelesenen Formel gebildet. Hierfür wird die Formel zunächst in das von HyKey-GUI beim Export benutzte Zwischenformat bestehend aus `HybridExpression`s übersetzt. Darauf wird dann der in 2.4 beschriebene Algorithmus angewandt.

4 Mögliche Erweiterungen

Im Rahmen des Projekts konnte nur die notwendige Funktionalität zur Benutzung von HySat-GUI als Oberfläche zur Eingabe von hybriden Automaten zur Benutzung mit KeYmaera geschaffen werden. In diesem Kapitel wird daher darauf eingegangen, welche zukünftigen Erweiterungen helfen können, HySat-GUI in diesem Zusammenhang zu verbessern. Es wird zudem kurz erläutert, wie deren Implementierung aussehen könnte, damit zukünftige Arbeiten an HySat-GUI vereinfacht werden.

Bevor weitere Änderungen an HySat-GUI vorgenommen werden, sollte überlegt werden, ob vorher eine *Refaktorisierung* von Teilen des Programms nötig ist. Bei einer Refaktorisierung werden bestehende Probleme an der Struktur eines Programms behoben, ohne die Funktion des Programms zu verändern. In Bezug auf HySat-GUI wäre dies sinnvoll, da es einige Schwächen aufweist. Beispielsweise gibt es in der Klasse `Main` sehr viele Abhängigkeiten, die eine Erweiterung des Benutzerinterfaces umständlich machen und teilweise schon jetzt unnötig sind. Zudem leidet die Verständlichkeit des Exports darunter, dass für das Zwischenmodell die gleichen Klassen genutzt werden wie für das Ausgangsmodell. Weiterhin ist die Nutzung der Klasse `Statics` in Hinsicht auf das Design fragwürdig. Es sollte untersucht werden, ob die dort gespeicherten Informationen für die Übersetzung direkt an die benötigten Stellen als Argumente übergeben werden sollten. Insgesamt könnten durch eine Refaktorisierung weitere Änderungen und Ergänzungen an HySat-GUI vereinfacht werden.

KeYmaera erlaubt es *Funktionen* zu definieren. Eine weitere Erweiterung wäre es, dies auch in HySat-GUI zu ermöglichen. Dafür muss der Parser für die Deklaration des hybriden Systems angepasst werden, um die Eingabe zu erlauben. Weiterhin müssen die deklarierten Funktionen im Modell gespeichert und im Benutzerinterface angezeigt werden. Letzendlich muss der Export nach KeYmaera so angepasst werden, dass die Funktionen ausgegeben werden.

Sync Label werden genutzt, um eine Synchronisation zwischen zwei oder mehr hybriden Automaten zu erzielen. Dazu werden Transitionen den verschiedenen Automaten mit einem gleichnamigen Label versehen. Anschließend dürfen die Transitionenen nur noch dann genutzt werden, wenn in allen Automaten jeweils eine Transition mit diesem Label zur gleichen Zeit genutzt wird. Die Implementierung dieses Features ist von besonderem Interesse, da die Nutzung von Sync Labels im PHAVer Format zwingend erforderlich ist. Um Sync Labels zu realisieren müssen neben dem Modell in Form der Klasse `Transition` auch noch der Eingabedialog und die von ihm generierten Commands, sämtliche Imports und Exports sowie das Speichern und Laden angepasst werden. Dafür müssen auch die entsprechenden Parser im Paket `parser` erweitert werden.

Eine weitere mögliche Erweiterung ist eine *Optimierung* des beim Export erzeugten Codes. Dadurch könnte die anschließende Verarbeitung mit KeYmaera beschleunigt werden. Da die Auswirkungen der Optimierung anwendungsspezifisch sind, muss die Optimierung unter Berücksichtigung der Eigenschaften von KeYmaera erfolgen. Eine

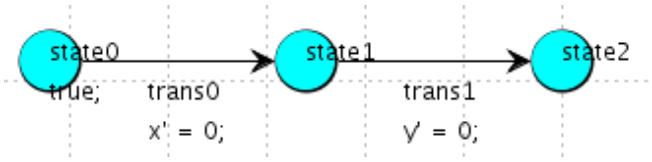

Abbildung 3: Unoptimierter Automat

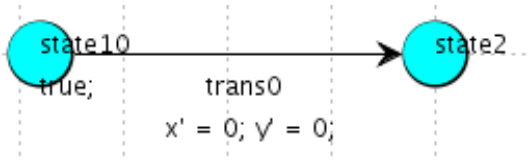

Abbildung 4: Optimierter Automat

mögliche Optimierung wäre es, zwei durch nur eine Transition verbundenen Zuständen durch eine sequenzielle Komposition anstatt durch eine nichtdeterministische Auswahl zu übersetzen.

Die beiden Zustände in Abbildung 3 könnten bei einer Optimierung zu denen in Abbildung 4 zusammengefasst werden, wodurch sich die Übersetzung von zwei nichtdeterministischen Auswahlen auf eine reduzieren würde. Dadurch könnte die Übersetzung von Listing 3 auf Listing 4 verkürzt werden, ohne die Semantik des hybriden Systems zu ändern.

Listing 3: Unoptimierter Code

```
1     (?(state=0);
2       (
3           (?(true); untitled0_variable_x := 0;  state := 1)
4         ++
5           ({true, true})
6       )
7     )
8   ++
9     (?(state=1);
10      (
11          (?(true); untitled0_variable_y := 0;  state := 2)
12        ++
13          ({true, true})
14      )
15    )
16  ++
17    (?(state=2);
18      (
19          ({true, true})
20      )
21    )
```

Listing 4: Optimierter Code

```
1      (?(state=0);
2        (
3              (?(true); untitled0_variable_x := 0; untitled0_variable_y := 0;
   state := 1)
4          ++
5              ({true, true})
6        )
7      )
8    ++
9      (?(state=1);
10       (
11             ({true, true})
12       )
13     )
```

Die Implementation der Optimierung könnte auf dem beim Export erzeugen Zwischenmodell erfolgen. Dadurch könnten die Änderungen am Rest der Übersetzung minimiert werden.

HySat-GUI kann außerdem ohne größeren Schwierigkeiten um neue Exports bzw. Imports erweitert werden. Dadurch könnte die Anzahl an bereits implementierten Modellen erhöht werden, welche mit KeYmaera benutzt werden können. Zusätzlich sind an der Benutzeroberfläche noch Verbesserungen möglich. So könnte beispielsweise eine Methode zur automatischen Anordnung von Zuständen und Transitionen implementiert werden, die Ausgabe von Fehlern bei der Übsetzung könnte verbessert werden oder es könnten weitere Optionen zur Anpassung des Programms an den Benutzer über das Menü zugänglich gemacht werden. Denkbar wären unter anderem veränderbare Tastenkombinationen. Weiterhin sollte bei einer weiteren Bearbeitung von HySat-GUI die Hilfe aktualisiert werden.

5 Fazit

Die wichtigsten Ziele des Projekts wurden erreicht. Es wurde wie gefordert eine vollständige Undo und Redo Funktionalität implementiert. Auch der Export nach KeYmaera wurde implementiert.
Bei der Implementation des Im- und Exports von PHAVer ist allerdings anzumerken, dass die Synchronisationslabel fehlen. Die Implementation der Label ist für die Benutzung von HySat-GUI als Werkzeug zur Eingabe von hybriden Programmen für KeYmaera zwar nicht unbedingt notwendig, in PHAVer sind Label allerdings verbindlich, wodurch beim Import Informationen verloren gehen.

Literatur

[1] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[3] Christian Schmitt. Graphische Benutzeroberfläche für HySAT. Individuelles Projekt an der Universität Oldenburg, 2007.

[4] A. Platzer. Differential logic for reasoning about hybrid systems. In A. Bemporad, A. Bicchi, and G. Buttazzo, editors, *Hybrid Systems: Computation and Control, 10th International Conference, HSCC 2007, Pisa, Italy, Proceedings*, volume 4416 of *LNCS*, pages 746–749. Springer-Verlag, 2007. `http://www.springer.com/comp/lncs/index.html`(c) Springer-Verlag.

[5] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[6] Goran Frehse. Language overview for phaver version 0.35, 2006.

[7] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

Hiermit versichere ich, die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Oldenburg, 08.02.2008

_____