Technische Universität Hamburg-Harburg

Bachelor Thesis

On the Application of Model-Checking Tools for Hybrid Automata

Jan Holste

July 19, 2012

supervised by:
Prof. Dr. Ralf Möller

Hamburg University of Technology
Institute for Software Systems
Schwarzenbergstrae 95
21073 Hamburg

Contents

1 Introduction

The differential dynamic logic ($\mathsf{d}\mathcal{L}$) is a logic for hybrid systems, introduced by Andre Platzer [2].

Hybrid Systems are systems, which are based on a continuous and a discrete behavior. The continuous behavior is often a physical influence of the environment. With hybrid automata we can model such systems.

A hybrid automaton is a non-deterministic finite state machine with differential equations in the states. That means, the execution of an automaton can stay for a finite time in a state and the current model behavior of this automaton can be changed in a specified way by differential equations. It is non-deterministic, whether an automaton stays in a specific state or take a transition to another state. With the help of conditions a specific range can be defined, in which the automaton can be executed. This is helpful to control the possible execution runs of an hybrid automaton.

Hybrid automata can be used to check and model real world problems. A project of Andre Platzer is the European Train Control System (ETCS) [4]. The ETCS is a rail track control system. It should allocate rail track blocks dynamically. In general, a train is only able to enter a rail track block if its free. The definition of such blocks is fixed, due to signals. With ETCS these blocks should be dynamic. The trains define their speed limit by the distance between them. To ensure that every train stops before it enters a dynamic block of another train, a secure gap is added. With the help of a model and a proof method, we are able to verify that specific security conditions hold for all possible execution runs. Therefore the $\mathsf{d}\mathcal{L}$ proof theorem [2] is introduced. Furthermore a program called KeYmaera [3] implements this theorem proof.

A common problem of model-checking and theorem proving is the runtime. Often the runtime of checking programs is exponential. In this Bachelor thesis, we take a closer look at KeYmaera [3] to analyze how it and the proof mechanism behind it works. Therefore, we introduce the basics of this topic in Chapter 2. This includes the differential dynamic logic ($\mathsf{d}\mathcal{L}$), the proof theorem and KeYmaera.

To analyze the runtime behavior of KeYmaera I build a program generator to generate test cases. Due to the high complexity of hybrid programs, I had to make some restrictions in the algorithms of the program generator. The restrictions and the algorithms are discussed in Chapter 3.

In Chapter 3 we also take a look at the test results. We try to find a function to predict the runtime of theorem proofs by hybrid automata indicators, for instance the number of states. We discuss how the restrictions of the generated programs manipulate the results and whether the generated programs behave like real world problems. Therefore we take an example and compare the runtime of it with the predicted runtime of the function we have fit.

Last but not least we summarize and come to a conclusion.

2 Dynamic Differential Logic d$\mathcal{L}$

In this section we introduce d$\mathcal{L}$ [2]. First of all we introduce the syntax of d$\mathcal{L}$ and based on the syntax the semantics. To get an understanding of d$\mathcal{L}$, we give an example for each part.

2.1 Syntax

The d$\mathcal{L}$ describes the differential and the discrete part of hybrid systems. To make restrictions with the help of conditions, whether a given discrete transition can be made or the execution could stay in a state for a finite time, we have to introduce first order formulas, see section 2.2 in [2].

2.1.1 First Order Logic

Terms are constructed by the signature $\sum$ and the set V of logical variables. The variables should be interpreted as reals. The $\sum$ contains real-valued functions like $0, 1, +, -, *, /$ and predicate symbols like $=,<,\leq,\geq,>$. With these few functions, we can construct terms like $a^3 < 6$. This would look like this: $a*a*a < 1+1+1+1+1+1$.

The set of all terms is $\mathrm{Trm}(\sum, V)$. It is defined as a minimal set, such that every variable of the set V is also an element of the set of all terms. Furthermore the set contains functions with at least one parameter. The parameters have to be in the set of all terms.

The set of formulas of first order logic is $Fml_{FOL}(\sum, V)$. It contains a predicate symbol of $\sum$, if and only if its parameters are terms which are in $Trm(\sum, V)$. If we have two formulas ϕ and ψ, then $\neg\phi$,$(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$ are also formulas. Furthermore if the formula ϕ is in $Fml_{FOL}(\sum, V)$ and a variable v is in V, then the all quantifier $(\forall x)$ and the exists quantifier $(\exists x)$ of x in respect to the given formula ϕ is also a formula.

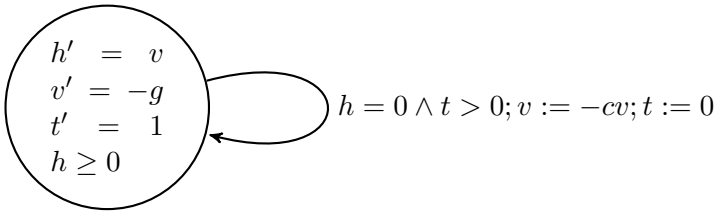

Figure 2.1: Bouncing Ball as Hybrid Automaton

2.1.2 Bouncing Ball Example

The bouncing ball example in Figure 2.1 has 5 variables. The variable h represents the current distance of the bouncing ball to the ground. The variable v is the speed of the ball. The speed is negative, if the ball moves towards the ground. If it bounces off the ground the speed is positive. In this example the time is explicitly expressed by the variable t. The example has only one state, which describes the movement of the ball. The hybrid automaton is non-deterministic, therefore the execution can stay in this state or take the reflexive transition. With the help of the condition $h \leq 0$ the execution can only run in this state as long as this condition is true. Such a condition in a state is called domain environment. This restriction models, that the ground is not higher then the ball. The reflexive transition has a condition, too. Only if the condition is true, the transition can be taken. In this case time t has to be bigger than zero and the distance has to be zero. To model that the bouncing ball can reduce the maximal distance with every jump, we introduce a variable c, which is from zero up to one. This value is chosen randomly.

The bouncing ball example has discrete and continuous assignments. The ordinary differential equations in a state models the continuous behavior. They are called continuous evolution of a state. The assignments of a transitions, also called jump sets, are discrete. The assignments of a continuous evolution or jump set are executed simultaneously.

2.1.3 Hybrid Programs

The minimal set of hybrid programs is called HP($\sum$,V), see definition 2.3 [2]. In the following part we describe statements, which are hybrid programs. A discrete jump set $(x_1 := \theta_1, ..., x_n := \theta_n)$ is a hybrid program, if x_i is in $\sum$ and θ_i is in Trm($\sum$,V) for $1 \leq i \leq n$. Furthermore all variables $x_1, .., x_n$ are different to each other. A continuous evolution $(x_1' := \theta_1, ..., x_n' := \theta_n \& \chi)$ contains ordinary differential equations and an evolution domain χ. Every ordinary differential equation $x_i' = \theta_i$ has to satisfy the following properties:

- x_i is in $\sum$
- θ_i is in Trm($\sum$,V)
- x_i' is the time derivative of x_i

If this holds and χ is a first order formula, then the continuous jump set is a hybrid program. $(?\chi)$ is a condition, where χ represents a first order formula. If this is the case, it is a hybrid program. If we have two hybrid programs α and β, then the combination with the non-deterministic operator $\cup$ and the concatenation with the operator ; are also a hybrid program. $(\alpha \cup \beta)$ means that one of them is non-deterministically chosen to be executed. $\alpha;\beta$ means that at first α will be executed and afterwards β. The loop of a hybrid program (α^*) is a hybrid program, too.

The operators of hybrid programs are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$, $[\alpha]\phi$, $\langle\alpha\rangle\phi$, $\exists x\phi$, $\forall x\phi$ and $p(\theta_1, ...\theta_n)$, where p is a predicate. [.] is the box operator. $[\alpha]\phi$ means, that for every run of the hybrid program α the first order formula ϕ has to hold. The diamond operator is $\langle.\rangle$. $\langle\alpha\rangle\phi$ is true, if and only if there exists a run of the hybrid program α, in which the first order formula ϕ holds.

2.1.4 Hybrid Program of Bouncing Ball

The hybrid program of the bouncing ball example [3] can be written down as shown in Figure 2.2. With the equivalent symbol $\equiv$, we declare the program name. In this example we call the program "ball".

$$
\begin{array}{l}
ball \equiv (\\
\qquad h' = v, v' = -g, t' = 1 \& h \geq 0; \\
\qquad (\\
\qquad\qquad ?(h = 0 \wedge t > 0); \\
\qquad\qquad (c' = 1 \cup c' = -1); \\
\qquad\qquad ?(0 \leq c < 1); \\
\qquad\qquad v := -cv; t := 0 \\
\qquad)\cup ?(h \neq 0 \vee t \leq 0) \\
)*
\end{array}
$$

Figure 2.2: Bouncing Ball Example as Hybrid Program

2.2 Semantics

In the last section, we introduced the first order and the differential dynamic formulas. In this section we introduce the semantics of these formulas, to be able to valuate them, see section 2.3 in [2].

2.2.1 Valuation of Terms

In $\mathsf{d}\mathcal{L}$, we have three different types of terms. The first type is the type of terms with rigid symbols in $\sum$. Their values are constant, for instance, $0, 1, +$ and $*$. The second type is terms with flexible symbols, which are also in $\sum$. Their value is changing over time, depending in which state the execution of the hybrid system currently is. And the last type is terms with logical variables, which do not change their value over time, but can be quantified with help of the existence and universal quantifier.

We have to interpret all of these symbols. Therefore we introduce an interpretation I. It interprets the predicates and functions as usual. For instance $I(+)$ represents the addition. Furthermore, we have an assignment function $\eta : V \to \mathbb{R}$ for logical variables. An assignment of x with a value $d \in \mathbb{R}$ is denoted by $\eta[x \to d]$. The set $\sum_{fl}$ represents all changeable state variables. The set of states is represented by $Sta(\sum)$. Each state v is represented by the mapping $v : \sum_{fl} \to \mathbb{R}$. The variable of $\sum_{fl}$ can have another value in another state. With the term state, we describe the possible execution states, in other words, all possible combinations of variable values and not the states of the hybrid automaton representation.

To interpret the current state of the hybrid system, we introduce the function val, which interprets a term with respect to the current state and logical variables. We denote $val_{I,\eta}(v, t)$, where I is the interpretation, η the mapping function of logical variables to their real values and v the current state of the hybrid system. This evaluates the term t. We have three different terms. If the term is a logical variable x, we call the function $\eta(x)$ to evaluate it. A state variable a is evaluated as a term by the function $V(a)$. If the term is a rigid function symbol of arity $n \geq 0$, we interpret the function f with $I(f)$ and the values with the help of η. This results in $I(f)(val_{I,\eta}(v, \theta_1), ..., val_{I,\eta}(v, \theta_n))$.

2.2.2 Valuation of Formulas

In this section we extend the definition of $val_{I,\eta}(v,t)$ for formulas of hybrid programs. The valuation of a formula is true, if and only if the formula for the given state of the hybrid system is true. For instance $val_{I,\eta}(v, \phi \vee \psi)$ is true, if and only if the valuation of ϕ or ψ is true with respect to I, η and the current state v. The valuation of $val_{I,\eta}(v, \forall x \phi)$ is true, if and only if the valuation of $val_{I,\eta[x \to d]}(v, \phi)$ is true for all $d \in \mathbb{R}$. The existence quantifier ($\exists$) is treated the same way.

All state transitions of a hybrid system are in the set $\rho_{I,\eta}(\alpha)$. The valuation of $val_{I,\eta}(v, [\alpha]\phi)$ is true, if and only if for all states ω $val_{I,\eta}(\omega, \phi)$ is true. That means, for all states of the hybrid program, ϕ has to hold, if the states are reachable. The same has to hold for the diamond operator $\langle . \rangle$.

2.2.3 Transitions

In $\mathsf{d}\mathcal{L}$, we have two different types of transitions. The continuous and the discrete transitions. In this section, we introduce the semantics of transitions.

A discrete transition (v, ω) is in the set of $\rho_{I,\eta}(x_1 := \theta_1, ..., x_n := \theta_n)$, if and only if all variables changed by the assignment $v[x_i \to val_{I,\eta}(v, \theta_i)]$ have the same values as the variables in the state ω. Of course, not all variables are changed by this transition. Some variables will not be touched. The continuous transition set $\rho_{I,\eta}(x_1' := \theta_1, ..., x_n' := \theta_n \& \chi)$ contains a transition (v, ω), if and only if there exists an evolution with the function f of time r, where χ is true for all states given by $f(i), i \in [0, r]$. Furthermore $f(0) = v$ and $f(r) = \omega$. Therefore f has to be differentiable in $(0, r)$. Some variables will not be touched by this transition and remain constant. The check of a condition χ ($?\chi$) does not change the state, but is a reflexive transition to the state itself. The transition of $(\alpha \cup \beta)$ is the non-deterministic choice of the two transitions α or β. Both hybrid programs are transitions from a state v to ω. The concatenation of two hybrid programs is the sequential execution of thus programs. The first program has to jump to a state, where the second program starts from. A loop is in the set of transitions, if and only if the end state can be reached by a finite repeat of a hybrid program, which starts at the start state. Furthermore, each transition in this loop has to be in the set of ρ.

The main idea behind $\mathsf{d}\mathcal{L}$ was introduced by this section. Now, we know the syntax and semantic of such programs. To get a more detailed view see Section 2.2 and 2.3 [2]. In the next section we will introduce, how the theorem proof is done. Furthermore we prove the given example, the bouncing ball, against a security condition.

$$(\wedge \text{ r})\ \frac{\vdash \phi \qquad \vdash \psi}{\vdash \phi \wedge \psi} \qquad (\rightarrow \text{ r})\ \frac{\phi \vdash \psi}{\vdash \phi \rightarrow \psi} \qquad (\text{ax})\ \frac{*}{\phi \vdash \phi}$$

$$(\wedge \text{ l})\ \frac{\phi, \psi \vdash}{\phi \wedge \psi \vdash} \qquad (\exists \text{ r})\ \frac{\vdash \phi(X)}{\vdash \exists \phi(x)}$$

$$(\text{i}\exists)\ \frac{\vdash QE(\exists X \wedge_i (\phi_i \vdash \psi_i))}{\phi_1 \vdash \psi_1, \ldots, \phi_n \vdash \psi_n} \qquad (\langle := \rangle)\ \frac{\phi_{x_1}^{\theta_1}, \ldots, \phi_{x_n}^{\theta_n}}{\langle x_1 := \theta_1, \ldots, x_n := \theta_n \rangle \phi}$$

$$(\langle' \rangle)\ \frac{\exists t \geq 0((\forall 0 \leq \tilde{t} \leq t \langle S_{\tilde{t}} \rangle \chi) \wedge \langle S_{\tilde{t}} \rangle \phi}{\langle x_1' = \theta_1, \ldots, x_2' = \theta_n \& \chi \rangle \phi}$$

Figure 2.3: A subset of the $\mathsf{d}\mathcal{L}$ rules

2.3 Proof Calculus

In this chapter, we take a short look at the proof calculus. The proof calculus is shown in section 2.5 [2] and uses the Gentzen-style sequent calculus. We introduce the Gentzen-style Sequent Calculus and the main idea of the proof theorem in this section.

2.3.1 Gentzen-style Sequent Calculus

The form of the sequent is written down as $\Gamma \vdash \Delta$. We denote the antecedent as Γ and the succedent Δ. The semantics of this construct is $\wedge_{\phi \in \Gamma} \phi \rightarrow \vee_{\psi \in \Delta} \psi$. The sequent symbol is interpreted as implication. The left side of the sequent is always in conjunctive form. The $\wedge$ is replaced by commas. The right side of the sequent is always in disjunctive form. The $\vee$ are in the same way replaced as the $\wedge$ by commas. Therefore it is important on which side of the sequent a term is mentioned.

The main idea of the proof is to retrieve an axiom at all branches. An axiom is a sequent $\Gamma, \phi \vdash \phi, \Delta$. This axiom is always true, because the left side is true, if and only if Γ and ϕ are true. The fact that ϕ is also on the right side implies a tautology. To get to the axioms, we have to apply rules to the base case.

The proof calculus contains a lot of rules to prove a $\mathsf{d}\mathcal{L}$ program. We want to introduce only a few and get the main idea of the proof calculus. To get the whole idea behind the proof including the quantifier elimination and the whole set of rules, take a look at section 2.5 [2].

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{*}{v \geq 0, z < m \vdash v^2 > 2b(m-z)}
}{\vdash v \geq 0 \wedge z < m \rightarrow v^2 > 2b(m-z)}
\rightarrow r, \wedge l
}{v \geq 0, z < m \vdash -\frac{b}{2}T^2 + vT + z > m}
}{v \geq 0, z < m \vdash \langle z := \frac{b}{2}T^2 + vT + z \rangle z > m}
\langle := \rangle
\qquad
i\exists \dfrac{*}{v \geq 0, z < m \vdash T \geq 0}
}{v \geq 0, z < m \vdash T \geq 0 \wedge \langle z := \frac{b}{2}T^2 + vT + z \rangle z > m}
\wedge r
}{v \geq 0, z < m \ \vdash \ \exists t \geq 0 \langle z := \frac{b}{2}T^2 + vT + z \rangle z > m}
\exists r
}{v \geq 0, z < m \ \vdash \ \langle z' = v, v' = -b \rangle z > m}
\langle' \rangle
}{\vdash \ v \geq 0 \wedge z < m \langle z' = v, v' = -b \rangle z > m}
\rightarrow r, \wedge l
$$

Figure 2.4: Application of proof rules

2.3.2 Proof Calculus Rules

In this section we introduce some rules of the proof calculus. Figure 2.3 show the rules, which we need for the proof example in the next section. The first rule we introduce is $(\wedge r)$. If we have $\vdash \phi$, then we have to show that both $\vdash \phi$ and $\vdash \psi$ is true. With this rule, the proof is cut into 2 branches, which can be handled separately. The rule $(\rightarrow$ r) is only a rewrite. $\vdash \phi \rightarrow \psi$ can transformed to $\vdash \neg\phi \vee \psi$. Due to the semantics of $\vdash$, we can move the ϕ to the left side. The result is $\phi \vdash \psi$. The rule (ax) and $(\wedge$ l) should be clear. (ax) is always a tautology and $(\wedge$ l) is the transformation, which was discuss in the section before. The rule $(\langle' \rangle)$ changes the diamond operator into a differential solution, which has to hold for a run of ϕ. The rule $(\langle := \rangle)$ transforms the assumption that there is a run such ϕ holds into a specific run, in which this holds.

2.3.3 Proof Example

Let $(v \geq 0 \wedge z < m)$ be the preconditions of the hybrid program $(z' = v, v' = -b)$. We want to prove whether the equation $v \geq 0 \wedge z < m \rightarrow \langle z' = v, v' = -b \rangle z > m$ always holds. This equation means that, if the preconditions are true, we assume that there is a run, in which the postcondition $z > m$ holds. There has to be only one run, because we use the diamond operator $\langle \rangle$. The proof is shown in Figure 2.4. We use the introduced rules to prove this. For a more detailed description see section 2.5 [2].

2.4 KeYmaera

KeYmaera [3] is a theorem proofer for hybrid systems and is free to use under the GNU General Public License. It implements the $\mathsf{d}\mathcal{L}$ theorem proof. The program is written in Java and uses the KeY-Theorem Prover as a basis. Some parts of the software are C-binaries to improve the runtime.

The structure of KeYmaera is a plug-in system. It applies the introduced $\mathsf{d}\mathcal{L}$ rules to the hybrid program and lets a solver calculate the result. The solvers are implemented as plug-ins. This enables a quick change of a solver. Every solver has a different behavior. The most common solvers are Mathematica 8 and Orbit. There are a lot of settings, which influence the success of a proof. We will show the used settings in our test cases.

3 Program Generation & Evaluation

A interesting aspect of the hybrid program proof is its runtime. In this chapter, we want to analyze the runtime behavior of KeYmaera. Therefore we build a program generator, which produces hybrid programs.

To generate programs, which can be compared to each other, we generate hybrid automata. That assures a strict control flow structure of generated hybrid programs. We are comparing hybrid automata in terms of the number of states, the number of discrete transitions, the number of ordinary differential equations and the number of evolution domain equations. Afterwards, we translate the automata into hybrid programs and extend them with pre- and postconditions.

We assume that two hybrid programs with equal number of states, discrete transitions, ordinary differential equations, pre- and postconditions have the same complexity and should have the same runtime.

In our case we want to check, whether we can predict the proof time with a prediction function, which we try to find from test cases.

3.1 Precondition Runtime Tests

The runtime behavior of a proof, should change, if we increase the number of elements like states, transitions or conditions. To get a first impression, we want to change one kind of element of the hybrid programs and let the others stay constant.

In this section we are creating an algorithm to generate hybrid programs, which represent hybrid automata with one state and a ordinary differential equation $a' = b$. The attribute we want to change is the number of preconditions. Therefore the generator should generate n preconditions and one postcondition. We generate only one postcondition to increase the possibility of a proved hybrid program.

3.1.1 Algorithm for Precondition Generation

In this section we introduce three algorithms. The first one generates terms and is called generateTerm (Algorithm 1). The second one uses the algorithm generateTerms to generate conditions. This algorithm we call generateConditions (Algorithm 2). The last algorithm (Algorithm 3) create a hybrid program with a set of

generated conditions as preconditions, a postcondition and a constant state with a continuous transition. The created automata have no discrete transitions.

All generators use randomness to construct program elements, like the selection of a variable. For each randomly chosen program element, we define a probability. The same approach is taken for creating benchmark problems for the Random 3-SAT benchmark [1].

Algorithm 1 generateTerm

Require: $|vars| > 0 \wedge recursion \geq 0$
  if $recursion = 0$ **then**
    $x \leftarrow getRandomNumber$
    $p \leftarrow x \mod |vars|$
    return $vars[p]$
  else
    $x \leftarrow getRandomNumber \mod 6$
    $y \leftarrow getRandomNumber \mod recursion$
    $z \leftarrow getRandomNumber \mod recursion$
    if $x = 0$ **then**
      return $generateTerm(vars, 0)$
    else if $x = 1$ **then**
      return $generateTerm(vars, y) + generateTerm(vars, z)$
    else if $x = 2$ **then**
      return $generateTerm(vars, y) - generateTerm(vars, z)$
    else if $x = 3$ **then**
      return $generateTerm(vars, y) * generateTerm(vars, z)$
    else if $x = 4$ **then**
      return $generateTerm(vars, y)/generateTerm(vars, z)$
    else
      return $generateTerm(vars, y)^{generateTerm(vars,z)}$
    end if
  end if

The algorithm (Algorithm 1) generates a term. The recursion defines how long the term should be. A recursion of zero is always a variable. The variable is chosen randomly. If the recursion is higher than zero, the algorithm returns one of six possibilities randomly. One of them is to return a variable. The other return values are the operators $+, -, *, /, exp$. These operators are return with a term on each side. These terms are generated recursively. The given recursive factor of the algorithm give a max recursive call. Therefore the algorithm should have the possibility to stop earlier. This is achieved with two random variables y, z. y defines the recursive parameter of the left term. z the other one. Each of them is defined with the help

of a random number and the modulo of the current recursive value. This ensures that the next call has at least one recursion less than the current call and halts in a finite time.

Algorithm 2 generateConditions

Require: $|vars| > 0 \land n \geq 0 \land d \geq 0$
  if $n = 0$ **then**
    $x \leftarrow getRandom \mod 5$
    if $x = 0$ **then**
      return $generateTerm(vars, d) = generateTerm(vars, d)$
    else if $x = 1$ **then**
      return $generateTerm(vars, d) > generateTerm(vars, d)$
    else if $x = 2$ **then**
      return $generateTerm(vars, d) < generateTerm(vars, d)$
    else if $x = 3$ **then**
      return $generateTerm(vars, d) \geq generateTerm(vars, d)$
    else
      return $generateTerm(vars, d) \leq generateTerm(vars, d)$
    end if
  else
    return $generateConditions(vars, 0, d)) \land generateConditions(vars, n-1, d))$
  end if

The generateConditions algorithm (Algorithm 2) generates conditions with 5 different operators. The operators are $<, >, \leq, \geq, =$. This choice is done by the modulo 5 of a random number. The algorithm generates n conditions, which generateTerm call has a maximal recursive depth of d.

Algorithm 3 generatePreConditonTest

Require: $vars = \{a, b, c, d, e, f, g, h, i, j\} \land pre = \#preconditions - 1 \land pre \geq 0 \land d = 1$
  $pres \leftarrow generateConditions(vars, pre - 1, d)$
  $posts \leftarrow generateConditions(vars, 0, 1)$
  $conTransitions \leftarrow (?q = 0; a' := b)$
  $disTransitions \leftarrow empty$
  return $hybridProgram(pres, posts, conTransitions, disTransitions)$

The last algorithm (Algorithm 3) generates a hybrid program, which represents an automaton with 10 variables $\{a, \ldots, j\}$, a number of preconditions and a condition recursion of 1.

3.1.2 Improvement of Precondition Tests

We obtain that the generated hybrid programs are often not provable. This could have a lot of issues. First of all, we want to discuss, what a hybrid program has to look like to be valid and improve the generation afterwards.

A hybrid program is proved, if and only if the implication

$$preconditions \rightarrow [hybrid\ program]postcondition$$

is validated. This could be the case, if the precondition were always false, due to contradictions in the preconditions like $a < 0 \land a > 0$.

A second case would be that all runs of the hybrid program fit the postcondition in respect to the preconditions. This is often not the case, because we are creating the pre- and postconditions randomly.

A contradiction would be reached quite fast, because we are increasing the state variable a with the derivative b in the state of the hybrid program. A postcondition with bounds a to a maximal value would invalidate the proof.

The hybrid program changes only the value of variable a. A proof of a hybrid program would be invalid too, if the pre- and postconditions contradicted.

To increase the possibility for a proved test, we choose a postcondition out of the preconditions randomly. This reduces the probability that the postcondition contradicts the preconditions.

3.1.3 Precondition Test Results

We generate tests with i preconditions where $1 \leq i \leq 5 \lor (10 \leq i \leq 50 \land i \mod 10 = 0) \lor (100 \leq i \leq 1000 \land i \mod 50 = 0) \lor (1100 \leq i \leq 2000 \land i \mod 100 = 0)$. Of the preconditions we take the first one as postcondition.

We prove these tests with KeYmaera using Orbital as solver. The settings of KeYmaera are in table 3.1 shown.

The tests are performed on a server with 4 AMD Opteron 6136 processors with 2,4 GHz and each test using RAM limited to 32 Gb. The whole RAM limit is 128 Gb. Due to the influence of runtime on cpu cache and disk access, only 2 tests run at the same time.

Settings	value
Differential Saturation	auto
First-Order Strategy	lazy
Counterexample	on
Real arithmetic solver	CohenHormander
Equation solver	Orbital
Differential equations	Orbital
Counterexample tool	SMT
Arithmetic simplifier	Orbital
built-in arithmetic	off
built-in inequalities	off
counterexample history	off
Strategy	DL Strategy

Table 3.1: KeYmaera settings for Orbital

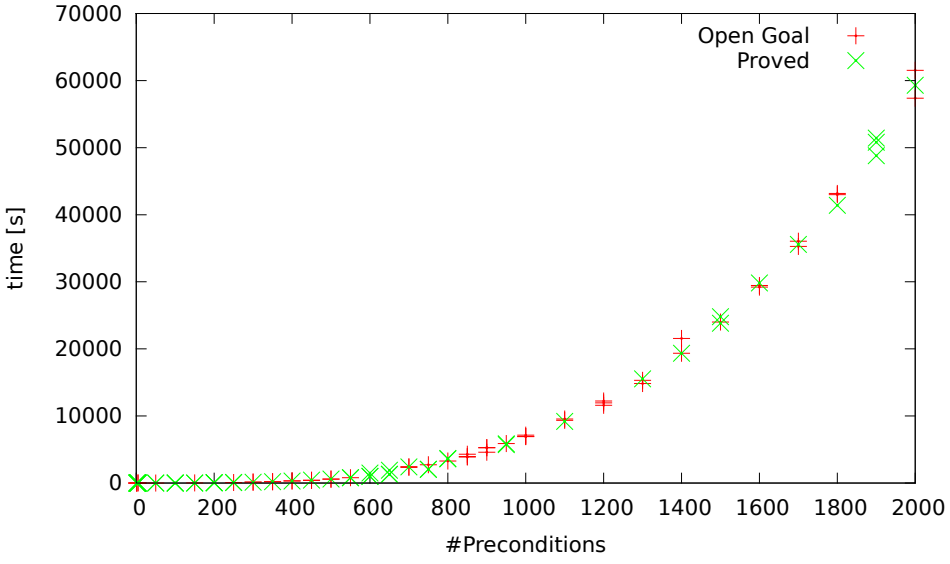

Figure 3.1: Comparison between open goal and proved

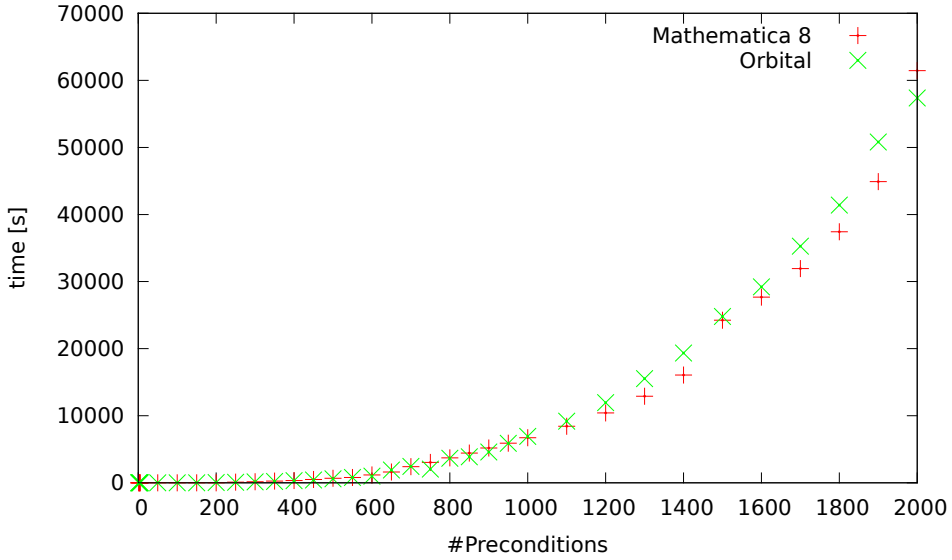

Figure 3.2: Comparison between Mathematica 8 and Orbital

From the Figure 3.1 we can obtain, that the runtime of the preconditions test is polynomial or exponential with respect to the number of preconditions. We assume that the runtime is exponential, because the SAT-Problem is in NP-Complete. A proof has to check whether at least the precondition is consistent. If this is not the case, the postconditions have to hold. Therefore it is reduced to a SAT-Problem.

The comparison of Mathematica 8 and Orbital in Figure 3.2 is done with the same tests. The tests that use Mathematica as solver use the settings of table 3.2. The tests with Orbital use 3.1. We obtain that each test needs almost the same runtime with both solvers. Orbital is a little bit slower. Furthermore Orbital implements the exponential function with a variable as exponent. This is not supported by Mathematica 8, or the KeYmaera plug in for Mathematica 8.

3.2 Hybrid Program Runtime Tests

In the previous section, we have obtained a runtime behavior that exponentially depends on the number of preconditions. In this section we want to create a complete hybrid program. We want to create hybrid automata, which are different in the number of states and differential equations. The number of discrete transitions

Settings	value
Differential Saturation	auto
First-Order Strategy	lazy
Counterexample	on
Real arithmetic solver	Mathematica
Equation solver	Mathematica
Differential equations	Mathematica
Counterexample tool	Mathematica
Arithmetic simplifier	Mathematica
built-in arithmetic	off
built-in inequalities	off
counterexample history	off
Strategy	DL Strategy

Table 3.2: KeYmaera settings for Mathematica

is $s * s * 0.2$. This is a random value, which fits the number of transitions in real systems quite well. With this fixed transition in comparison to the number of states, a complete graph should be avoided. An automaton with 2 states, but 10 transitions is quite unrealistic. Therefore we use the round up of $2 * 2 * 0.2$. The result is 1 transition.

Furthermore we set the number of evolution domain equations to the number of states. In the average every state has one evolution domain equation. The number of preconditions is not fixed. It should behave in a flexible way like the test before. The automaton should have one postcondition, too. As was done in the last test, we use the first precondition as a postcondition.

To decrease the complexity of the condition equations, we generate only equations with a maximal depth of 0. This means that we will generate equations with only one variable as term on each side of the comparison operator. This would for instance be $a < b$.

3.2.1 Algorithm for Hybrid Program Generation

In this section, we want to introduce the used algorithms and explain how they should be interpreted. The function "getRandomNumber" is a helper function, which returns a uniformly distributed random number. The second helper function "getRandomNumberList" returns a list of uniformly distributed random numbers.

The first algorithm (Algorithm 4) is an algorithm, which creates a list of tuples. A tuple includes the number of differential equations and evolution domain equations of a state. Later, we want to generate states with the help of the list of tuples, which

Algorithm 4 generateStateNumbers

Require: $n > 0 \wedge con \geq 0 \wedge evo \geq 0$
  $cons \leftarrow getRandomNumberList\ of\ length\ con$
  $evos \leftarrow getRandomNumberList\ of\ length\ evo$
  $statesNumbers \leftarrow List\ of\ initial\ states\ of\ length\ n$
  for all $k \in cons$ **do**
    $statesNumbers[k \mod states]\ increase\ \#continuous$
  end for
  for all $k \in evos$ **do**
    $statesNumbers[k \mod states]\ increase\ \#evolutiondomain$
  end for
  return $statesNumbers$

indicates the facts of a state. We generate hybrid automata with a fixed number of states, differential equations and evolution domain equations. The number of these indicators are for the whole hybrid automata. Therefore we need an algorithm, which assigns the number of differential and evolution domain equations to each state. Algorithm 4 achieves this task.

The algorithm has the number of states, differential equations and evolution domain equations as parameters. The algorithm initializes a list of n tuples. Each tuple represents a state with no equations. With a list of n uniformly distributed random numbers, we select tuples to increase in the first step the number of differential equations of the states. In the second step we take another list of uniformly distributed random numbers to increase the number of evolution domain equations. At the end, the algorithm returns a frequency of differential and evolution domain equations over the states.

I decided to achieve the frequency in this way, to be able to set the number of different equations before the frequency is done. Therefore I construct automata, which have the same properties in terms of number of states, differential and evolution domain equations. If I would randomly generate the equations, I would not achieve a uniformly distributed frequency over all possible numbers of states for only a few tests. With this algorithm I can create automata with fixed properties. This helps me a lot, because I test the automata manually.

Algorithm 5 is quite simple. It generates differential equations, which have either a variable or a number as derivative. The number is an integer value from 0 up to 10. Which case and which variables or number are used, is decided by random numbers. As parameter, the algorithm takes a list of variables.

The algorithm to generate Assignments (Algorithm 6) is quite the similar to algorithm 2. The only difference is, that the operator is an assignment and not a

Algorithm 5 generateContinuous

Require: $|vars| > 0$
  $x \leftarrow getRandomNumber \mod 2$
  $y \leftarrow getRandomNumber \mod |vars|$
  $z \leftarrow getRandomNumber$
  if $x = 2$ **then**
    return $(vars[y])' = vars[z \mod |vars|]$
  else
    return $(vars[y])' = (z \mod 10)$
  end if

Algorithm 6 generateAssigment

Require: $|vars| > 0$
  $x \leftarrow getRandomNumber \mod 2$
  $y \leftarrow getRandomNumber \mod |vars|$
  $z \leftarrow getRandomNumber$
  if $x = 2$ **then**
    return $(vars[y]) := vars[z \mod |vars|]$
  else
    return $(vars[y]) := (z \mod 10)$
  end if

Algorithm 7 generateState

Require: $|vars| > 0 \wedge con \geq 0 \wedge evo \geq 0$
  $continuous \leftarrow \{\}$
  $evolutiondomain \leftarrow \{\}$
  for $k \in con$ **do**
    $continuous \leftarrow \{continous, generateContinous(vars)\}$
  end for
  $evolutiondomain \leftarrow generateConditions(vars, evo, 0)$
  return $state(continuous, evolutiondomain)$

derivative. The generateState algorithm (Algorithm 7), which should be applied to all tuples of the generateStateNumbers, generates a state with the given number of differential equations and evolution domain equations. We say that a state has an evolution domain, if and only if it has at least one evolution domain equation. The algorithm generates the evolution domain equations with the help of the generateConditions algorithm (Algorithm 2). The first parameter is the list of variables, the second one is the number of conditions and the last one is the depth of the rules. We use the depth of 0, which means only conditions with one term of a variable of each side of the equation operator. For instance $a < b$. The continuous equations are generated with the help of the generateContinous algorithm.

Algorithm 8 generateDiscrete

Require: $|vars| > 0, |states| > 0$
  $p \leftarrow getRandomNumber \mod 2$
  $q \leftarrow getRandomNumber \mod 2$
  $r \leftarrow getRandomNumber \mod |vars|$
  $pre \leftarrow \{\}$
  $ass \leftarrow \{\}$
  $post \leftarrow \{\}$
  if $p = 0$ **then**
    $pre \leftarrow generateConditions(vars, 0, 0)$
  end if
  $k = 0$
  while $k < r$ **do**
    $ass \leftarrow \{ass, generateAssigment(vars)\}$
    $k = k + 1$
  end while
  if $q = 0$ **then**
    $post \leftarrow generateConditions(vars, 0, 0)$
  end if
  return $dis(pre, ass, post)$

The generateDiscrete algorithm (Algorithm 8) generates discrete transitions. A discrete transition has a precondition, which has to hold for it to appear in the execution path. A discrete transition can have a lot of preconditions. We are generating only transitions with up to one precondition. The same holds for the postconditions. The transitions also has up to one postcondition. The postconditions have to hold after the assignments have been made. The number of assignments is taken randomly and 0 to $|vars|$ assignments can be generated for the transition.

The generateHybridProgram algorithm (Algorithm 9) generates hybrid programs with given parameters in terms of the number of states, the number of differential

Algorithm 9 generateHybridProgram

Require: $s > 0 \wedge c > 0 \wedge pr > 0$
  $vars \leftarrow a, \ldots, j$
  $states \leftarrow \{\}$
  $dis \leftarrow \{\}$
  for all $(con, evo) \in generateStateNumbers(s, c, s)$ **do**
    $states \leftarrow \{states, generateState(vars, con, evo)\}$
  end for
  for all $i \in \{1, \ldots, (s * s)/5\}$ **do**
    $dis \leftarrow \{states, generateDiscrete(vars, states)\}$
  end for
  $pre \leftarrow generateConditions(vars, pr, 0)$
  $post \leftarrow pre[0]$
  return $hybridProgram(vars, pre, states, dis, post)$

equations in the whole hybrid automaton and the number of preconditions. We use the introduced algorithms. We define the list of variables as a set $\{a, \ldots, j\}$. We then generate a list of s tuples with the help of the generateStateNumbers algorithm, where s is the number of states. Furthermore we pass the number of differential equations as variable c. The number of evolution domain equations should be equal to the number of states. Therefore, we pass the number of states variables twice as a parameter of generateStateNumbers.

We generate the precondition the same way as in the test before. We use the generateConditions algorithm to generate them. As postcondition we take the first precondition. This should increase the chance that the hybrid program is proved.

3.2.2 Improvement of Hybrid Program Tests

We want to generate a three dimensional test field. The first dimension is the number of states, which should be from 1 up to 200. The second dimension represents the number of differential equations in the automaton. We want to use 1 up to 200 differential equations. The last dimension represents the number of preconditions, which is between 1 and 10. We choose only 10 as a maximal value, because we expect that in real world examples, automata with more preconditions don't appear frequently. In fact, the number of states and continuous equations are also quite high for hybrid programs.

Only 2 of the first 20 hybrid programs are proved. The number of proved hybrid programs is very low. To understand the problems, we have to understand when a hybrid program is not proved. First of all, a program is always true, if the preconditions are contradicted. This is always the case in the precondition tests with a much

higher number of preconditions. Furthermore, the preconditions of the precondition tests are more complex in terms of the depth. We defined depth in the algorithm for generating such conditions. This means that only a few contradictions appear within ten preconditions.

We have clarified that the proof in general doesn't have a problem with the preconditions. The problem has to be in the automata with their differential equations and the discrete preconditions. Due to the random generation, we generate states with differential equations that are not limited by an evolution domain, because there is none or the variables which are involved are not in the differential equations.

To generate a hybrid program which tries to keep a look at this problem would be too general. It would then act like a model checker. With this assumption we would generate hybrid programs with much more side effects then we have now. Yet we only have conditions with variables and assignments, derivatives with variables and numbers from 0 up to 10. This does not reflects the real world.

To improve the test results we have to make sure that every automaton is proved to be true. This can be achieved by using a variable in the postcondition, which does not change in the hybrid program. Therefore we generate the hybrid programs in the same way as before and replace the first precondition and the postcondition with $zz > 0$. This ensures that every automaton is proved to true, because the precondition is contradicted or the postcondition is true for all runs. The second statement, that the postcondition is true for all runs is trivial. Therefore we assume that each automaton is true.

3.2.3 Hybrid Program Test Results

In this chapter, we want to generate programs with the given algorithms of the previous section. We generate programs with $1, 20, 40, 60, 80, 100$ states. For each number of states, we generate programs with $1, 20, 40, 80, 100, 120, 140, 160, 180, 200$ differential equations respectively. A discrete transition has the number of variables as assignments as maximum by definition. In this case the programs have ten variables $\{a, \ldots, j\}$. Each program has ten generated preconditions. The first precondition is replaced by $zz > 0$, which is also the postcondition. For all possible runs this postcondition has to hold. We can observe that this is true for all generated programs.

For these tests we use the same hardware and KeYmaera settings 3.2 for Mathematica as in the precondition tests.

The result of the tests is a three-dimensional graph. The first dimension are the states. The second one are the differential equations. And the last one is the runtime

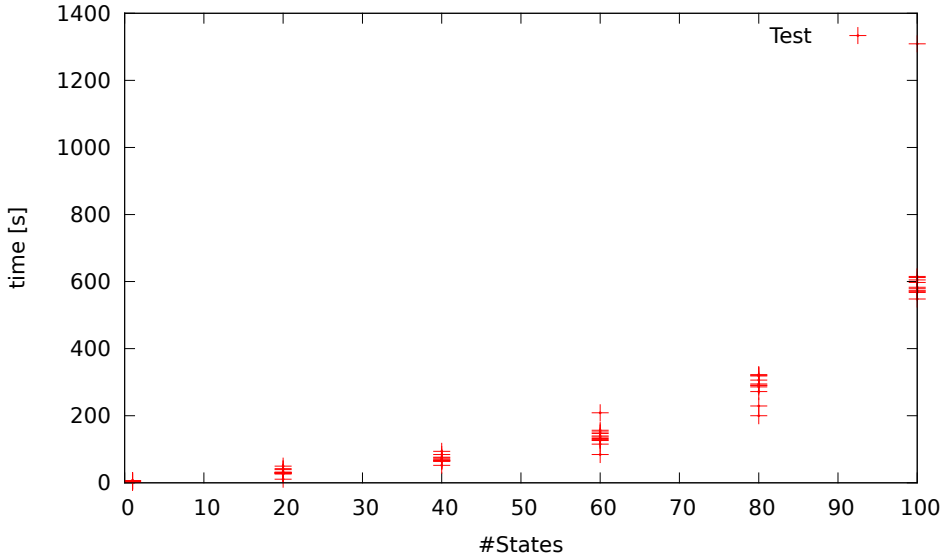

Figure 3.3: Comparison of runtime and number of states

in seconds. The differential equation is only one equation. We define the continuous evolution as a set of equations. Therefore a lot of the differential equations can appear in the same state.

We run the tests on the same configuration with Mathematica 8 as the precondition tests before.

Due to the fact that the graph would be crowded, we try to get a clearer view on the result set and take a look only at Figure 3.3, which represents the results in two dimensions. The first one is the number of states, the second one the runtime in seconds. For each different number of states, there are eleven test cases, because we have eleven combinations of number of differential equations.

We obtain that the variation of the runtime of a state number value is quite small. This variation of runtime can appear due to a different number of differential equations. To satisfy this assumption we take a look at the next figure, which represents the number of differential equations against the runtime in seconds. The figure shows the results for each number of states in another color. This gives us the opportunity to interpret the influence of the number of differential equations on runtime behavior.

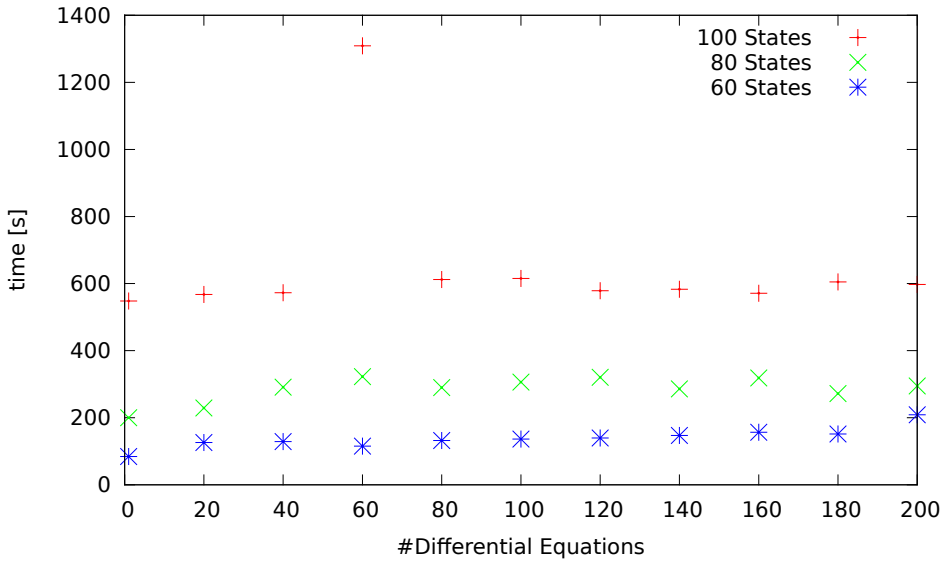

Figure 3.4: Comparision of runtime and number of differential equations

In Figure 3.4 a comparison of the runtime in terms of the number of differential equations is shown. We selected all tests with 100,80 and 60 states. In summary each of the different state numbers have 11 tests from 1 up to 200. As we can obtain from this figure, the differential equations have an impact on the runtime. The runtime for all numbers of differential equations is changed. The runtime makes big jumps depending on the number of states. The variance of the runtime in respect to the number of differential equations is much smaller. I would assume that the runtime of a hybrid program increases, if the number of differential equations increases. This might be not the case, as seen in this figure. We know that each pair of hybrid programs, which are compared, have the same number of states. Furthermore we have a fixed number of discrete transitions and a maximum of ten assignments for each transition. Therefore only the number of assignments and the position of discrete and continuous equations are changed. The term position is the source state and destination state of a transition. For a continuous evolution both states are the same.

The graph of an automaton with n states and k discrete transitions can be drawn in different ways. The number of directed edges of a complete directed graph is:

$$number\ of\ edges\ = \binom{n}{2}$$

The formula for k out of m chosen discrete transitions, with the opportunity to choose a transition more than once is:

$$combinations\ of\ transitions = \binom{k+m-1}{k}$$

For a hybrid program, which represents a hybrid automaton with 10 states and therefore $10^2/5 = 20$ discrete transitions, the number possible transitions is:

$$possible\ transitions = \binom{20}{2} = 190$$

To choose 20 discrete transitions out of the 190 possible ones, also with the possibility of choosing one more then once, the number of combinations is:

$$number\ of\ graphs = \binom{190+20-1}{20} = 4.9638 \times 10^{27}$$

This number indicates how many different hybrid programs we can generate. This is only the number of different hybrid automata in terms of discrete transitions. We did not consider the possible combinations of assignments on the discrete transitions with pre- and postconditions. These different automata have to be different in their runtime by assumption. This might be the case and is expressed in the figure above.

Nevertheless we want to try to get a lower bound function of runtime with this small test set compared to the possible automata.

3.2.4 Runtime Prediction Function

In this section we want to try to fit the whole runtime into a function, which has the number of states and continuous equations as parameter. We assume that the the runtime is exponential in respect to the number of states. Therefore we try to fit the runtime of the states in following function $f(x)$:

$$f(x) = a * x^b + c$$

where x represents the number of states. We fit this function with the help of least squares regression and get the following function:

$$f(x) = 0.000299 \times x^{3.16446} + 0.99720$$

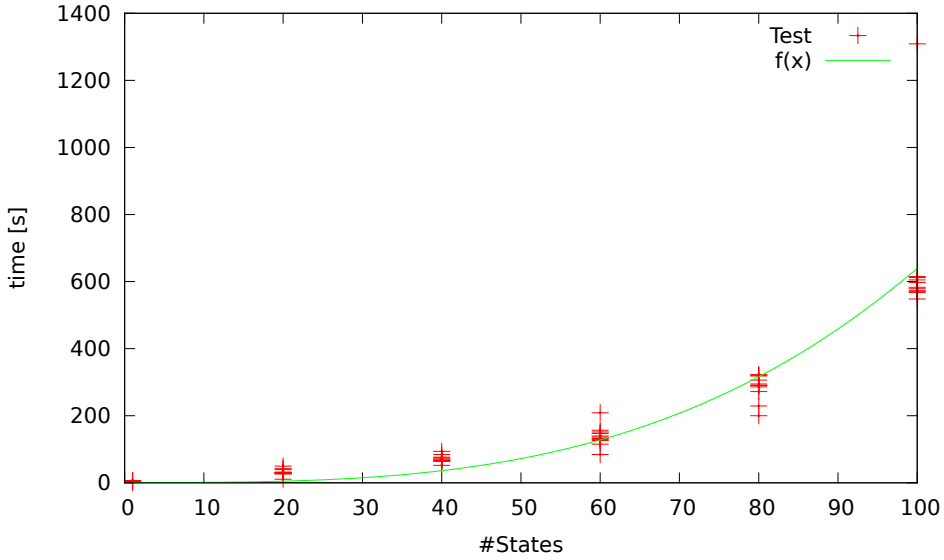

Figure 3.5: Comparision of fit function f(x) with test set

The variance of the runtime in terms of fixed state numbers and changing number of differential equations is non-deterministic. There is no possible fit, because the runtime is non-deterministic and has always another value. The runtime behavior seems to be not touched by the number of differential equations. The influence of the differential equations on the runtime is very small. We try to fit the runtime of hybrid programs only depending on the number of states. In Figure 3.6, we can see the function f(x) with the test results. We obtain that it doesn't describe the test results perfectly. We will check this function against an example in the next subsection.

3.2.5 Check against Example

In this section we check whether the function f(x), introduced in the previous section, predicts the right runtime for the bouncing ball example.

We want to check whether the bouncing ball example from Andre Platzer has a similar runtime. The bouncing ball has one state and one discrete transition. The transition has 2 assignments and 2 preconditions. The differential equations in the state are $h' = v, v' = g, t' = 1$ with the evolution domain $h \geq 0$. The discrete transition is $v := -c * v; t := 0$ with the precondition $?h > 0 \ \& \ t > 0$. h represents the current distance of the ball to the ground. v represents the current speed and g the gravitation. t stands for time. As preconditions we have $g > 0 \ \& \ 0 \leq c <$

$1 \;\&\; h = g/2 * t^2 + v * t \;\&\; h \geq 0 \;\&\; v \leq -g * t + V$. The idea of this hybrid program was presented in the introduction and should be clear. With the precondition we make sure that the ball has a positive distance from the ground to the position in the air. The gravitation should be greater than zero. Furthermore the physical behavior of the ball at the beginning of the hybrid program is set as the precondition $h = g/2 * t^2 + v * t$. With the help of the hybrid program we want to prove that every run satisfies the postcondition $h \geq 0$.

If we plug one state into the function $f(x)$, we get $f(1) = 0,99$. This result means, that we predict a runtime of about 1 second. The proof of the hybrid program takes 4.8 seconds. The question which arises is why the proof takes about 3.8 seconds more than a test case.

If we take a look at the fitted function, we see that the function is below the runtime results in the first part of the test cases in terms of the number of states. To get a more detailed view, we take a look on the results itself.

Time[s]	States	Preconditions	Differential Equations
5.6	1	10	1
5.8	1	10	20
5.7	1	10	40
5.4	1	10	60
6.3	1	10	80
2.4	1	10	100
5.9	1	10	120
2.0	1	10	140
6.1	1	10	160
5.6	1	10	180
2.4	1	10	200

Table 3.3: Test-Results

The arithmetic mean of the runtime results in the table 3.3 is 4.83 seconds. This value fits the runtime of the bouncing ball problem exactly.

The runtimes vary because of the system load of the server. The proof is done by Mathematica 8, which also has to be initialized first. This can sometimes cause an additional runtime of 5 seconds. The same proof after a complete restart of the program KeYmaera can have different runtimes. Therefore the time is not always the same and varies from time to time. This can be seen in the table. One test takes only 2.0 seconds, but has seven times more states than a test which takes 5.8 seconds. This also shows the complexity of the programs. It is quite difficult to model hybrid programs with the same complexity.

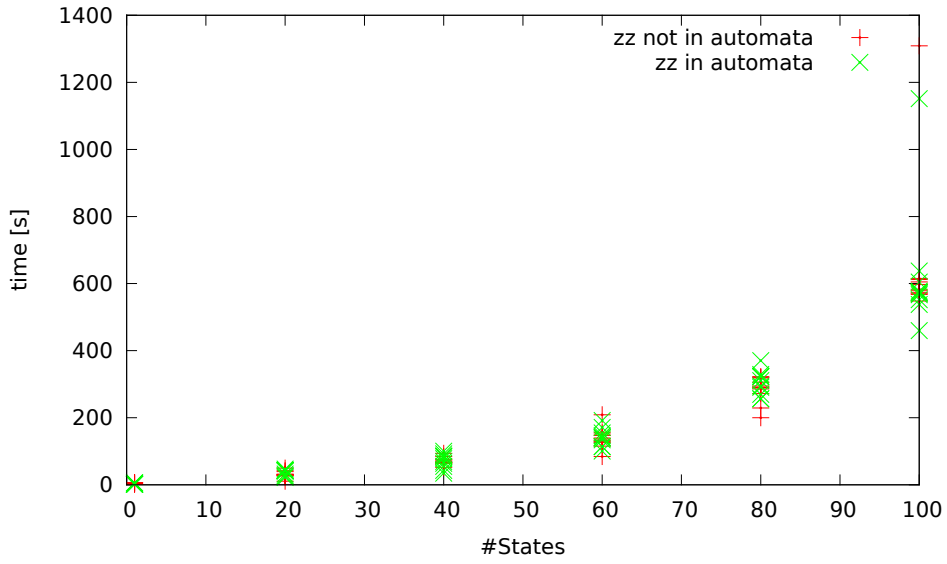

Figure 3.6: Comparision of automata with/without the variable zz in terms of states

3.2.6 Postcondition Variables in Tests

The postcondition $zz > 0$, which we checked for the test set of hybrid programs, included only one variable. This one variable is zz. It does not appear in the automaton itself, only in the precondition. Furthermore it is the same condition as in the postcondition. Therefore the question arises, whether the runtime of a proof with postconditions is faster, if the variables of the postcondition are not mentioned in the hybrid automaton. It could be possible that the theorem prover cuts some parts, because the variable is not mentioned in them.

To check whether this behavior occurs, we run a second test set, which also uses the variable zz for assignments. It is only assigned to other variables. Therefore the variable does not change over time and the postcondition $zz > 0$ holds for every execution state.

Figure 3.6 shows the tests with zz variables in the automata and without them, in respect to the number of states and the runtime. We obtain that both tests are in the same region. Only a few tests need a bit more time to validate. But on the

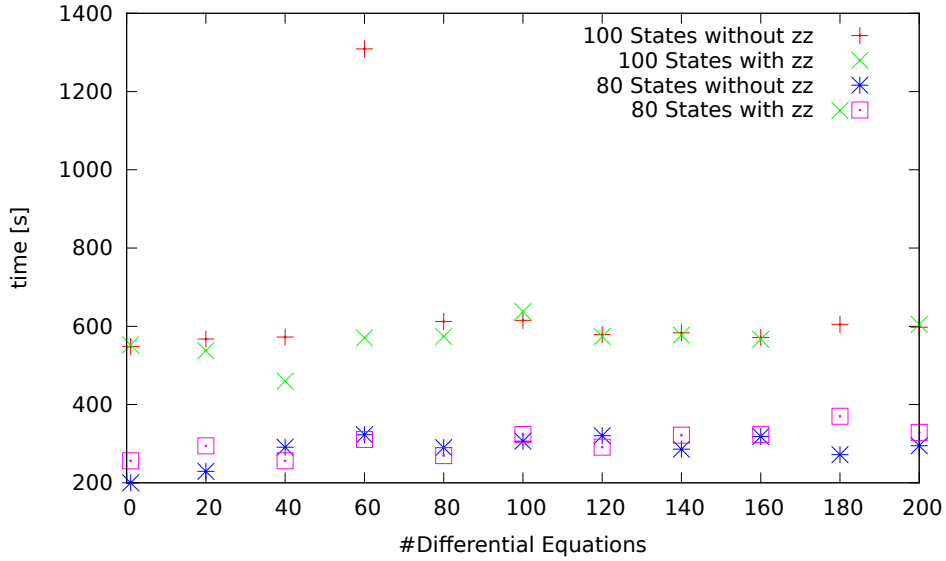

Figure 3.7: Comparision of automata with/without the variable zz in terms of differential equations

other hand there are a few tests, which need less time. The Figure 3.7 shows, like before, the number of differential equations in respect to the runtime.

For 100 states we have for each number of differential equations a test, in which zz only appear in the pre- and postconditions, and a test with zz as variable in the automaton. The number of occurrence is randomly. This repeats for the case that we have 80 states in the other two test rows which are in the bottom. If we look at both comparisons, we obtain that not only the figure before, but also in this one, both tests with same number of states have nearly the same runtime. Therefore we can assume that both tests need the same runtime to be validated.

The second figure shows that some tests are more complex than other ones. We mentioned before that there is a large number of combinations and each hybrid program can be differently in its complexity, even while having the same number of states and differential equations.

4 Conclusion

In this bachelor thesis, we built a generator for hybrid programs to analyze the runtime behavior of theorem proving. We discussed two different approaches. The first one was to look at the precondition evaluation. We obtained that the runtime increases depending on the number of preconditions in an exponential way.

The second approach was the generation of complete hybrid automata, which we transform to hybrid programs and add pre- and postconditions. We realize that it is quite difficult to compare two runs to each other, if they cannot be proved. This is because we do not know whether a counterexample was found or the whole program was checked. Therefore we generated only programs, which are always provable. We saw that we were able to fit a function in terms of the number of states, although it only describes a lower bound of the runtime. This was shown with the bouncing ball example.

All in all we can say that there are a lot of different hybrid programs, which represents hybrid automata with the same number of states, discrete transitions and differential equations. Each of them have different runtime and can be different in their complexity. Let a hybrid automaton have two states, a discrete transition from the start to the second state and two variables x_1, x_2 with the precondition $x_1 < x_2$. The automaton also has two differential equations $x_1' = 1, x_2' = 2$ in the start state and none in the second. The same hybrid automaton with a differential equation in each state is more complex to prove, if we check that x_1 have to be unequal to x_2 in all possible runs. Furthermore the proof results would be different. The first program is proved, the second fails.

Nevertheless, we have shown that a few programs, which are provable with KeYmaera and have easy conditions like the generated tests, have nearly the same runtime. This is only an observation on a very small test set, but gives us the impression that KeYmaera proves all postconditions of hybrid programs the same way.

Bibliography

[1] Fabio Massacci. Design and results of the tableaux-99 non-classical (modal) systems comparison. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX '99, pages 14–18, London, UK, UK, 1999. Springer-Verlag.

[2] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[3] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[4] André Platzer and Jan-David Quesel. European Train Control System: A case study in formal verification. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.