FORMAL VERIFICATION OF AN ELECTROMECHANICAL SYSTEM WITH DISCONTINUOUS PROPERTIES

A dissertation submitted to The University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2010

Paul Duckworth
School of Mathematics

Contents

List of Figures

List of Tables

Abstract

This project investigates a discontinuous dynamical example under the hybrid system framework. We simulate simple discontinuous systems to gain insight into how to treat surfaces of discontinuity within hybrid systems. We then simulate the main example of this project, the electromechanical system which is known to have catastrophic behaviour under certain conditions. We investigate the discontinuous system and the three types of behaviour it can exhibit.

To reduce unwanted behaviour within the system we turn our attention to the use of formal verification techniques. These can be used to check certain properties about the hybrid dynamical system in an automated way. It is possible to prove safety conditions about the system in software such as KeYmaera. We pass the electromechanical system into KeYmaera by representing it first as a hybrid system, then translating it into a hybrid program. This allows us to test and create safe control parameter values for the system without the need for simulation, which can be computationally expensive.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Paul Duckworth would like to thank his supervisor, Eva. M. Navarro-López for her continued support during this final project. He would also like to thank Len Freeman and Tony Shardlow, Course Directors of the Mathematics and Computational Science MSc he has been studying during the academic year 2009-10.

Chapter 1

Introduction

1.1 Motivation

Real life engineering and mathematical problems can be studied and understood using dynamical systems. They allow us to investigate and analyze the behaviour of a system, and study what factors affect it. The importance of modelling and simulating dynamical systems is huge, with many systems describing interesting real world problems which need to be clarified.

When a system combines continuous dynamics and discrete transitions it is referred to as discontinuous, or switched. This is because it exhibits switching behaviour between states due to discrete switch dynamics. These combined dynamics can be studied using hybrid systems theory, which combine the continuous and discrete dynamics into one model. There are many modelling frameworks for hybrid systems. The three main ones will be looked at in further detail later.

This project seeks to address formal verification in a class of hybrid systems. Formal verification techniques arose because of the need to prove correctness properties of computer programs and algorithms using formal methods. Recently however, these techniques have been mapped over to dynamical

systems theory. Here dynamical systems are passed in to verification software with the use of program notation. Properties of hybrid programs can be proved to show correctness statements of hybrid systems and to identify safety critical regions of parameters by finding reachable states under certain control parameters. This can allow for safe systems to be built without the need for simulation, which can be computationally expensive. Using a hybrid systems framework, we intend to explore the use of formal verification techniques on a discontinuous dynamical example.

Our example system is a two degrees of freedom, torsional model of a simplified conventional vertical oilwell drillstring specified in [20]. It is a particular case of the n-DOF model proposed in [24]. We first simulate the system to study its long term behaviours which are known to have catastrophic results under certain initial conditions. We then pass the system into formal verification software, KeYmaera. This will allow us to verify the system's dynamics given particular initial conditions. Finally, we use the results of the verification to set a range of safe initial conditions which will reduce any unwanted (and sometimes catastrophic) long term behaviour of the system.

1.2 Dynamical Systems

Dynamical systems theory is the mathematical theory of systems that change with time. Historically, the moving particles and bodies of classical mechanics led to the abstract idea of a dynamical system. Modern theory however can be used to describe many kinds of time evolving systems, including biological, ecological and economic, as well as those from the more conventional areas of physics and engineering.

A dynamical system has two main parts, there is a set of states often in $\mathbb{R}^n$ and a specification of how the states change over time. The description

of how the states change with time can be expressed in one of two ways: either a set of differential equations if time is continuous, or it can be a set of difference equations if time moves in unit steps. Common continuous dynamics come in the form of the differential equations specified by Newton's laws, an example of this type is given later. For further background theory to dynamical systems refer to [14].

1.3 Discontinuous Dynamical Systems

Discontinuous dynamical systems are systems that evolve continuously (smooth) until there is a discontinuity (jump or transition) in some part of the system. There are several types of discontinuous systems, depending on the type of discontinuity considered. We investigate an example of an impact system later: the case of a bouncing ball. The bouncing ball falls with continuous dynamics until it hits the floor (an impact) and then a discrete jump in the dynamics is present as the ball's velocity undergoes a discrete change: it changes direction and is reduced by some dampening factor. Discontinuous dynamical systems are often called non-smooth or switched systems.

The main example in this project is a discontinuous dynamical system which includes a discontinuous friction torque, and has one surface of discontinuity associated with angular velocity zero. We discuss the system more in Section 2.

Friction is a common discontinuity for mechanical systems as the friction a body experiences is often related to its current velocity, such that every time the velocity changes sign, the friction experienced changes sign. This change of sign however, would not create the complex dynamical behaviours or oscillations which occur. They arise from the property that the friction acting on a body trying to start moving is greater in magnitude than the friction acting on the body when it is already moving.

Indeed, the friction torque on the surface of discontinuity (zero angular velocity) is unknown and varies within a closed set. A major drawback of the discontinuous friction torque is that we need to have a model for the friction at the discontinuous point. For example we have a surface of discontinuity, Σ, and dynamics on that surface, $\dot{\boldsymbol{x}}$, which change depending on where the trajectory is with respect to the surface:

$$\dot{\boldsymbol{x}} = \begin{cases} f^+(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{X}^+, \\ f^-(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{X}^-. \end{cases} \tag{1.1}$$

The trajectory is in $\mathcal{X}^+$ if $\sigma(\boldsymbol{x}) > 0$ and in $\mathcal{X}^-$ if $\sigma(\boldsymbol{x}) < 0$, where σ is the smooth function $\sigma : \mathcal{X} \to \mathbb{R}$. The set:

$$\Sigma := \{\boldsymbol{x} \in \mathbb{R}^n : \sigma(x) = 0\},$$

is defined as the sliding manifold or switching surface. Therefore from Equation 1.1 the dynamics either side of the surface of discontinuity are defined but if $\boldsymbol{x} \in \Sigma$ i.e. $\sigma(x) = 0$, we don't know $\dot{\boldsymbol{x}}$. So we need a suitable model to obtain the dynamics when the trajectory is on this surface. This is all shown in Figure 1.1 below:

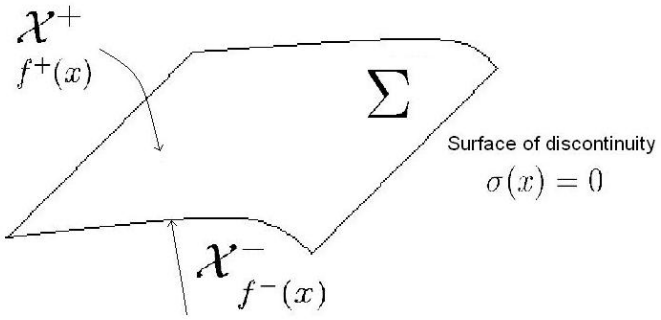

Figure 1.1: Surface of discontinuity Σ due to friction.

There are various methods that can be used to find a representation at the discontinuity. We proceed by describing the one we will use when studying

our main example. It is a method commonly used by control engineers, and is called Utkin's Equivalent Control Method [34].

Utkin's Equivalent Method

Utkin's Equivalent Control Method may be considered as a physical interpretation of the Filippov method [35]. The general idea is as follows.
Consider a discontinuous system of the form:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}), \qquad \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{u} \in \mathbb{R}^m.$$

The control function $\boldsymbol{u}$, can take one of two values, depending upon the sign of $\sigma(\boldsymbol{x})$:

$$\boldsymbol{u}(\boldsymbol{x}) = \begin{cases} \boldsymbol{u}(\boldsymbol{x})^+ & \text{if } \sigma(\boldsymbol{x}) > 0, \\ \boldsymbol{u}(\boldsymbol{x})^- & \text{if } \sigma(\boldsymbol{x}) < 0. \end{cases}$$

Utkin's method finds a vector field tangential to the trajectory by solving the equation:

$$\frac{\partial \sigma}{\partial \boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{u}) = 0, \tag{1.2}$$

so the dynamics on $\sigma(\boldsymbol{x}) = 0$ are governed by $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u_{eq}})$.

It is clear that if a trajectory visits the surface of discontinuity, it spends an amount of time on either side of it first, i.e. the system's dynamics are $\dot{\boldsymbol{x}} = f^+(\boldsymbol{x})$ for Δt_1 interval of time and $\dot{\boldsymbol{x}} = f^-(\boldsymbol{x})$ for Δt_2 interval of time, therefore the average dynamics during the total time, $\Delta t = \Delta t_1 + \Delta t_2$, is $\mu f^+(\boldsymbol{x}) + (1-\mu) f^-(\boldsymbol{x})$ for some constant μ.

The dynamics on the surface of discontinuity are then set to the average dynamics of the trajectory, which is specified by the formula of time spent on either side of the discontinuity, this has the expression:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \mu f^+(\boldsymbol{x}) + (1-\mu) f^-(\boldsymbol{x}) \\ &= f_0(\boldsymbol{x}). \end{aligned} \tag{1.3}$$

This can be thought of as the average dynamics:

$$\dot{\boldsymbol{x}}_{av} = \frac{\Delta x}{\Delta t} = \frac{f^+(\boldsymbol{x})\Delta t_1 + f^-(\boldsymbol{x})\Delta t_2}{\Delta t_1 + \Delta t_2} = \mu f^+(\boldsymbol{x}) + (1-\mu) f^-(\boldsymbol{x}), \qquad (1.4)$$

with the final equality holding because of Filippov's Method. From this and Equation 1.2 we obtain an expression for μ:

$$\begin{aligned} \dot{\sigma} &= \Delta\sigma(x)(\mu f^+(x) + (1-\mu) f^-(x)) = 0, \\ &\Rightarrow \mu = \tfrac{\Delta\sigma(x) f^-(x)}{\Delta\sigma(x)(f^-(x) - f^+(x))}. \end{aligned} \qquad (1.5)$$

The dynamics on the surface of discontinuity then become $\dot{\boldsymbol{x}} = f_0(\boldsymbol{x})$ with μ defined as in Equation 1.5. This is the method that we will use when a surface of discontinuity arises due to friction in our main example of a torsional vertical drillstring.

Further Points

There are three key analysis aspects which make modelling and simulating any discontinuous dynamical systems difficult and it is worth discussing them before we continue:

1. The systems usually exhibit a wide range of complex behaviours which often lead to faults, the dynamics affect the performance of the system in some way. These complex behaviours usually have their origin in what is known as a discontinuity-induced bifurcation (DIB). This is outside the scope of this project but refer to [5] for further information.

2. There is sometimes non-uniqueness or even non-existence of solutions when the trajectories of dynamical systems cross or slide on a discontinuity surface.

3. There are also simulation and numerical integration problems: zero-crossing location and detection. This can sometimes be overcome by using intelligent software.

For these reasons, dynamical systems (discontinuous or not) can be modelled and simulated under the hybrid system framework. This is a relatively new area of dynamical systems and aims to revolutionise the way we look at systems.

1.4 Hybrid Systems

A hybrid system is a system characterized by the coupling between continuous-type dynamics and discrete-event dynamics. For example, an automobile engine whose fuel injection (continuous) is regulated by a microprocessor (discrete). Systems like these arise when finite-state machines (discrete dynamics) accompany controllers modelled by partial or ordinary differential equations (continuous dynamics). They imply that trajectories evolve smoothly throughout a continuous state space, satisfying the differential equation until some condition is satisfied and a discrete-event is triggered inducing a change in the model characteristics.

Many real-world examples of dynamical systems can be reinterpreted under the framework of hybrid systems such as relays, switches and motion controllers. Other much larger examples, where extensive safety critical research has taken place, include automated highway systems [37] and flight control and collision avoidance manoeuvres [29]. Some applications of hybrid systems will be looked at in greater detail in Section 1.5.

There are several representations of hybrid dynamical systems, each has originated from, and is orientated around a specific type of problem. The main three hybrid dynamical modelling frameworks are distinguishable. First is a computer scientists approach; it is the automaton-based framework which merges continuous dynamics and finite automata theories [3, 12]. Second, the equation based and event-flow formulae representation [4, 8, 9, 32]; this is similar to the modelling frameworks for continuous systems, but with added

reset or transition functions to specify the discrete dynamics. Finally, we have the behavioural representation [36, 39]; where the systems are defined by specifying the behavior or time evolution in some manner. For example, by means of the set of trajectories on a metric space. We take a look at each of these before proceeding.

1.4.1 Hybrid Automata

As embedded computing becomes increasingly popular and important, hybrid systems are increasingly being used in safety critical applications; making correctness and reliability a major concern. For this purpose, the hybrid automaton was proposed as a formal model for hybrid systems [12].

Hybrid automata is a graph related representation which is similar to computer science discrete representations and symbolic dynamics. It is a way of representing a hybrid dynamical system in terms of a finite automaton combined with a state space model. This approach is the one most used by computer scientists and is the one we will proceed using. So lets first define a finite automaton [21].

Definition 1 (Finite automaton [13]) A finite automaton is a quintuple

$$A = (Q, \Sigma, \delta, q_0, F),$$

where:

- Q is a finite set of discrete states or *locations*.
- Σ is the finite set of input symbols or *events*.
- δ is the *transition function* that takes a state and an input symbol, and returns a state.
- q_0 is the initial state.
- F is the set of final or *accepting* states: $F \subset Q$.

Often depicted by a graph, a finite automaton has discrete locations represented by vertices, and transitions represented by edges. The elements of the set Σ can then be seen to be labels for the edges on the graph. An example of a three location finite automaton is shown below in Figure 1.2, where the three locations are specified by the set $Q = \{q_1, q_2, q_3\}$. In this example $q_0 = q_3$ from the definition, q_1 is the final state and the edges are labelled σ_i for $i = 1, \ldots 3$.

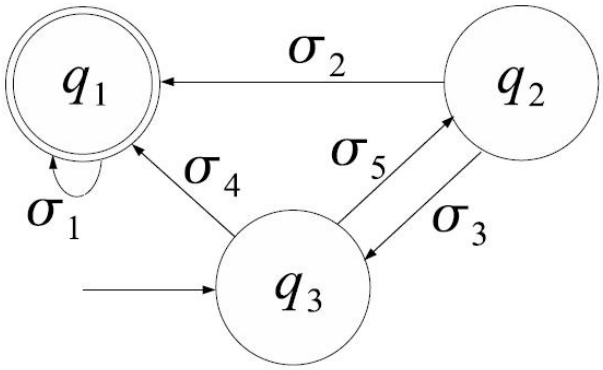

Figure 1.2: Finite automaton [10].

Extending this definition to apply for hybrid systems we must define a hybrid automaton. Proposed in [21, 22] is the notion that each discrete location has its own continuous dynamics and domain. This means we need to specify initial states, and continuous inputs and output spaces for each location, this causes the notation to get slightly trickier.

Definition 2 (Hybrid automaton [21, 22]) A hybrid automaton with inputs and outputs is a collection:

$$H = (Q, E, \mathcal{X}, \Sigma, \mathcal{U}, O, \mathcal{Y}, Dom, \mathcal{F}, Init, G, R, h, r),$$

where:

- Q $= \{q_1, q_2, \ldots, q_N\}$ is a finite set of locations.
- E $\subseteq Q \times Q$ is a finite set of edges called transitions or events.
- $\mathcal{X} \subseteq \mathbb{R}^n$ is the continuous state space.
- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_M\}$ is a finite set of symbols labelling the edges and representing the discrete input events.
- $\mathcal{U} \subseteq \mathbb{R}^m$ is the continuous input space.
- $O = \{o_1, o_2, \ldots, o_k\}$ is the finite set of symbols representing the discrete output events.
- $\mathcal{Y} \subseteq \mathbb{R}^m$ is the continuous output space.
- $Dom : Q \rightarrow 2^{\mathcal{X} \times \mathcal{U}}$ is the location domain. It is a mapping from the locations Q to the set of all subsets of $\mathcal{X} \times \mathcal{U}$, that is, Dom assigns a set of continuous states and inputs to each discrete state $q_i \in Q$, thus, $Dom(q_i) \subset \mathcal{X} \times \mathcal{U}$.
- $\mathcal{F} = \{\ \boldsymbol{f}_{q_i}(\boldsymbol{x}, \boldsymbol{u})\text{: } q_i \in Q\}$ is the collection of vector fields describing the continuous dynamics such that $\boldsymbol{f}_{q_i} : \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}$.
- $Init \subseteq Q \times \mathcal{X}$ is the set of initial states.
- $G : E \rightarrow 2^{\mathcal{X}}$ is a guard set. Function G assigns to each edge a set of continuous states: this set contains the states which enable transition along that edge.
- $R : E \times \mathcal{X} \times \mathcal{U} \rightarrow 2^{\mathcal{X}}$ is a reset map for the continuous states for each edge. It is assumed to be non-empty so that the dynamics cannot be destroyed, only changed.
- $h : Q \times \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{Y}$ is the continuous output mapping, there is one for each location.

- $r : Q \times \mathcal{X} \times \Sigma \times \mathcal{U} \rightarrow O$ is the discrete output map, there is one for each location.

Again, this definition can be seen in graph form, Figure 1.3 shows a 3 location automaton with specified continuous dynamics inside each of the states $\{q_1, q_2, q_3\}$.

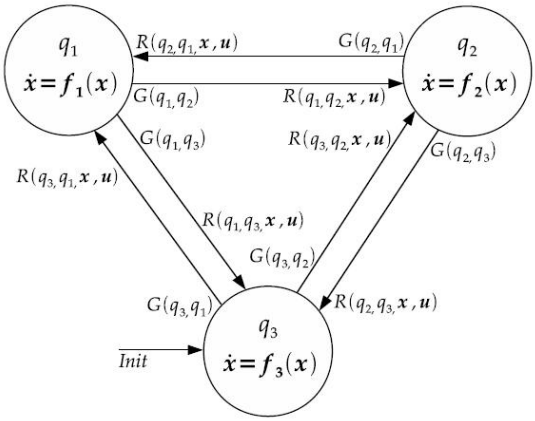

Figure 1.3: Hybrid automaton [10].

This definition of a hybrid automaton implies that two types of transition can occur. Externally induced transitions, which are known as controlled and secondly, internally induced transitions, known as autonomous.

The system evolves for a certain time under some dynamics specified by the system's current location. Autonomous transitions occur when a guard condition is satisfied by the continuous state vector. This enables the system to change to the location which that condition is guarding. The reset then applies, which re-initialises the continuous state to a specified value. Note that this can be the identity, so that no change actually occurs. The system then continues to evolve under the dynamics specified by the new location.

Figure 1.3 defines a system that is initialised in location q_3, and evolves

under the dynamics $\dot{\boldsymbol{x}} = f_3(\boldsymbol{x})$. This occurs until one of the two guard conditions associated with location q_3, either $G(q_3, q_1)$ or $G(q_3, q_2)$, are satisfied. If one of these transitions is made then the reset condition associated with the transition taken is applied, either $R(q_3, q_1, \boldsymbol{x}, \boldsymbol{u})$ or $R(q_3, q_2, \boldsymbol{x}, \boldsymbol{u})$. The system then continues to evolve in the new location, stimulated by the new continuous dynamics.

1.4.2 Event-Flow Representation

Simply put, hybrid dynamical systems studied under this framework are an indexed collection of ODEs along with a map for 'jumping' among them. The jumping occurs whenever the trajectories current state satisfies some criteria or conditions, given by its membership in a specified subset of the state space. Therefore the whole system can be thought of as a patching together of ODEs coupled with initial and final states [8].

Formally a hybrid dynamical system (HDS) under this framework is a system [8]:

$$H = (Q, \boldsymbol{\Sigma}, \boldsymbol{A}, \boldsymbol{G}),$$

with parts:

- Q is a countable set of discrete states.
- $\boldsymbol{\Sigma} = \{\Sigma_q\}_{q \in Q}$ is the collection of systems of ODEs. For each q, the vector field f_q, represents the continuous dynamics of the trajectory evolving.
- $\mathbf{A} = \{A_q\}_{q \in Q}$, is the collection of autonomous jump sets.
- $\mathbf{G} = \{G_q\}_{q \in Q}$, are the autonomous jump transition maps, these represent the discrete-event dynamics.

Therefore the dynamics of $H = (Q, \Sigma, A, G)$, can be summarised in the following way:

- The system is assumed to start in some hybrid state in the state space S excluding A, $S \backslash A$, say $s_0 = (x_0, q_0)$.
- It evolves according to $\dot{x} = f_{q_0}(x)$ until the state enters (if ever) A_{q_0}, at which point $s_1^- = (x_1^-, q_0)$.
- At this time it is instantly transferred according to the transition map to $G_{q_0}(x_1^-) = (x_1, q_1) \equiv s_1$, from which point the process continues.

We are not studying our discontinuous example under this framework so for further details about this approach, refer to [8].

1.4.3 Behavioural Representation

This approach is not commonly used, but the system is defined by specifying its behaviour or time evolution. It requires a large understanding of techniques used to fully describe, including precise definitions of Time Spaces, Equivalent Time Spaces, and Motion, all of which and more are defined in [39]. This approach leads to the hybrid dynamical system also being defined as a quintuple. Our discontinuous example is not studied under this framework either, so for further reading refer to [39].

1.5 Hybrid System Examples

There are many real-world examples of hybrid dynamical systems. They arise in all aspects of life where continuous dynamics and discrete-events describe a physical system. Some modern day hybrid systems are complex, and due to their importance, they often need their dynamics verified to find a safe region of their state space. We describe a few examples next, showing firstly how hybrid systems can be used in applications and secondly to get an understanding of how to model and simulate discontinuous dynamical systems.

Simple physical systems such as switches and relays can be modelled as hybrid systems, with their dynamics considered discontinuous given some event

i.e. a blown fuse. More interesting functions can be described as differential equations with some internal memory within the system, such systems exhibit path dependence. A classic example of which is a thermostatically controlled room.

1.5.1 Thermostatically Controlled Room

In this example a furnace is switched on or off depending on the current and desired room temperature, denoted by x and x_0 respectively. The situation can be depicted by a multi-valued function H shown in Figure 1.4.

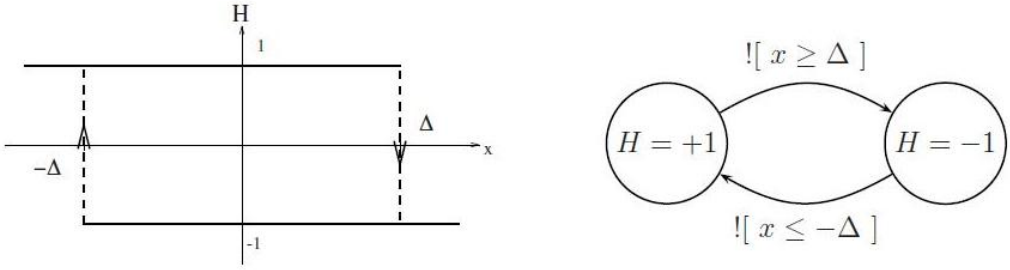

Figure 1.4: Hysteresis function, H, (left) and finite automaton associated with H (right) [8].

If we let the function H model the hysteretic behavior of a thermostat, then the thermostatically controlled room may be modelled as follows:

$$\dot{x} = f(x, \mathrm{H}(x - x_0), u).$$

The function f denotes the dynamics of temperature and whether the furnace is turned on or off, and u is some auxiliary control signal like burn rate.

1.5.2 Water-Tank Example

Another classic example of a real-world application is an automated water-tank. This particular example regulates water level, y, between 1 and 12

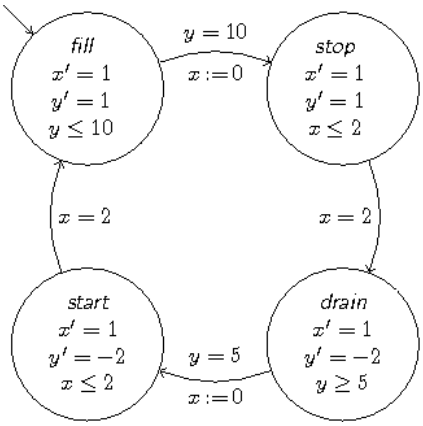

Figure 1.5: Hybrid automaton representation of a hybrid water-tank example [30].

units by filling or emptying a tank. Figure 1.5 is the corresponding 4-state hybrid automaton for the system with locations fill, stop, drain and start.

The system is initialized in the 'fill' state and evolves under the dynamics shown inside that location i.e. $\dot{x} = 1$ and $\dot{y} = 1$. Each location has a guard condition and reset to represent the discrete dynamics. For example, when the water level, y, becomes equal to 10 units, the system transits to the stop location and the clock variable, x, is reset to zero. Each state has a domain specified under the dynamics, this can be treated as an invariant for each location. An invariant is a condition which is always true whilst the system is in that location. This example was extracted from [30], refer to for further details.

1.5.3 Bouncing Ball Example

This third example we look at, that of a bouncing ball, is one which will greatly help with our understanding of our main example of a discontinuous system. As we go through this project we refer to and often compare to this

less complex discontinuous system. This is because it can be studied as a discontinuous dynamical system under the hybrid systems framework in a very similar way to our main example which is introduced in Section 2.

The hybrid automaton for a one state bouncing ball is given in Figure 1.6, the dynamics are specified inside the single location. The system variables include: h (the height of ball), v (the velocity of the ball), g (the acceleration due to gravity), and c (dampening constant). The notation, h' is the partial differential of height with respect to time, i.e. dh/dt.

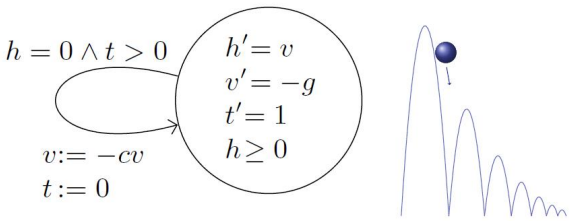

Figure 1.6: Hybrid automaton representation of a bouncing ball [30].

The dynamics of the system appear very simple, with height and velocity varying until the transition condition is satisfied. This occurs when height is equal to zero and time is greater than zero. At which time the reset conditions apply, the ball bounces so the velocity is reversed and the time is reset. However, even this simple discontinuous example is difficult to simulate due to the events associated with the impact of the ball and the floor.

Using the system state vector, $\boldsymbol{x} = (h(t), v(t))^T = (x_1, x_2)^T$ the equations of motion become:

$$\begin{aligned} \dot{x_1} &= v, \\ \dot{x_2} &= -g. \end{aligned}$$

The bouncing ball displays a jump in a continuous state (velocity) at the surface of discontinuity, or transition condition, $h = 0$:

$$v^+ = -c \times v^-,$$

where c is the dampening coefficient: $0 \leq c < 1$.

We passed this example into MatLab using naive subroutines, the code written can be seen in Appendix A.

Although this program makes perfect sense, the ODE solvers in MatLab can not solve the system because of the discontinuous surface at $x_1 = 0$. MatLab can not integrate the system's differential equations near this surface. It therefore fails to produce solutions and fails to simulate the system correctly.

Since this system is similar to our main example system, we searched for solutions to the problems associated with the surface of discontinuity. We found two: First, is to use MatLab intelligently and define an event based ODE solver, making use of *discrete event dynamics* to give the results of the system. An event based ODE solver for the bouncing ball is already programmed into MatLab as a piece of example code called *ballode.m*. It works because unlike our naively written MatLab code, it does not try to integrate the system of equations near the surface of discontinuity, $x_1 = 0$. The MathWork Inc. code can be seen in Appendix B.

This code runs a demo of a bouncing ball which is an example of repeated event location, where the initial conditions are changed after each terminal event (in this case after each bounce). The demo computes ten bounces with calls to the ODE23 solver. The speed of the ball is dampened by 0.9 after each bounce and the trajectory is plotted using the output function odeplot [31].

This code was ran using initial conditions, $(h(t), v(t)) = (20, 0)$, and is plotted in Figure 1.7. The events can be clearly seen as the red circles on the graph occurring when the ball bounces, i.e. when height = 0m.

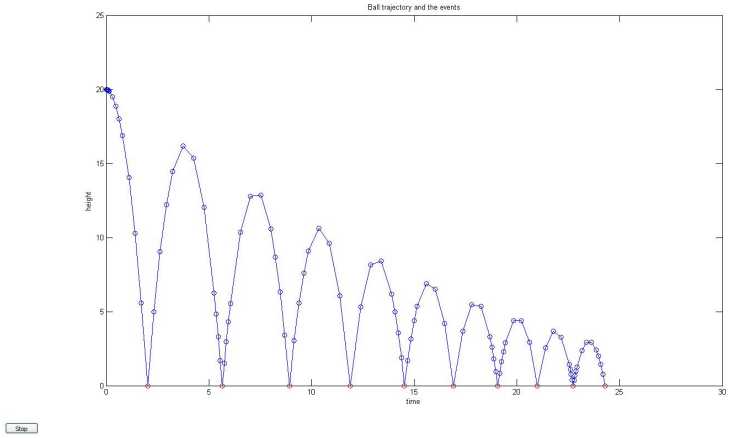

Figure 1.7: Event based solver: bouncing ball, simulated using *ballode.m*.

There is little improvements that can be made to the *ballode.m* MatLab code since it has been developed by MathWorks Inc. who have considerable knowledge in this field.

However, it uses the ODE23 solver which is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine [7]. We know the ODE45 is a similar solver but based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair [11]. They are both one-step solvers, for example, in computing $y(t_n)$ it only needs the one solution at the immediately preceding time point, $y(t_{n-1})$. At crude tolerances ODE23 can be more efficient than ODE45, but it is clear that to produce more accurate results the ODE23 solver could be replaced by the ODE45 solver, possibly at the expense of execution time.

The second solution to the problem of integrating near the surface of discontinuity was found via the easy-to-use MatLab package 'Simulink'. Developed by MathWorks Inc., Simulink is a tool specifically for modelling, simulating and analyzing dynamical systems. The bouncing ball example is common in Simulink, it makes use of the graphical block diagramming tool, using the block libraries, to simulate the system. Figure 1.8 shows how using two integrator blocks we can build the model of the bouncing ball in Simulink.

The Integrator on the left is the velocity integrator modelling the first equation and the Integrator on the right is the position integrator. The position integrator block dialog sets a lower limit of zero. This condition represents the constraint that the ball cannot go below the ground.

This Simulink example is extracted from the MathWork Inc. website [2] and uses initial conditions $(h_0, v_0) = (10, 15)$, and a dampening coefficient of 0.8.

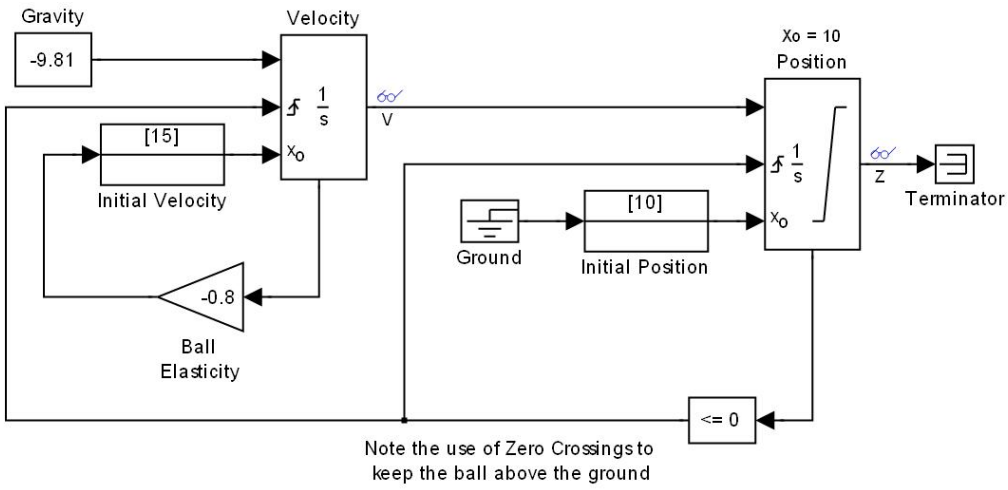

Figure 1.8: Bouncing ball Simulink model [2].

The analysis into a simple discontinuous dynamical system such as the bouncing ball has highlighted some of the difficulties we will face with our main discontinuous example.

There are many more examples of real-world applications of hybrid dynamical systems such as disk drives, hopping robot control, vehicle powertrains, automated highway systems and flight control. These are addressed in [8].

Chapter 2

System under Study: A Discontinuous System

In this section we describe the main example of this project: an electromechanical system outlined by Dr Eva M. Navarro-López in [19, 25]. It is a very interesting system, with discontinuous state derivatives (exhibiting sliding motion). This system exhibits different types of long term behaviours due to a surface of discontinuity associated with angular velocity zero. This system will later be used to present some verification results in the software KeYmaera. Before we can begin verification however, we must take a closer look at the system and it's dynamics.

The electromechanical system is a conventional vertical oilwell drillstring. It consists of a rotating mechanism at the surface, a set of vertical drill pipes, which are screwed one to each other to form a long pipeline, and the BHA (bottom-hole-assembly, also referred to as the 'bit'). We use a simplified 2 Degrees of Freedom model to adequately capture the most important dynamical properties, shown in Figure 2.1.

The drill pipes are represented by a linear spring which effectively models the torsional stiffness k_t and torsional damping c_t. This spring connects

the two rotary inertias, J_r and J_b, at the top rotary system (the surface) and bit respectively.

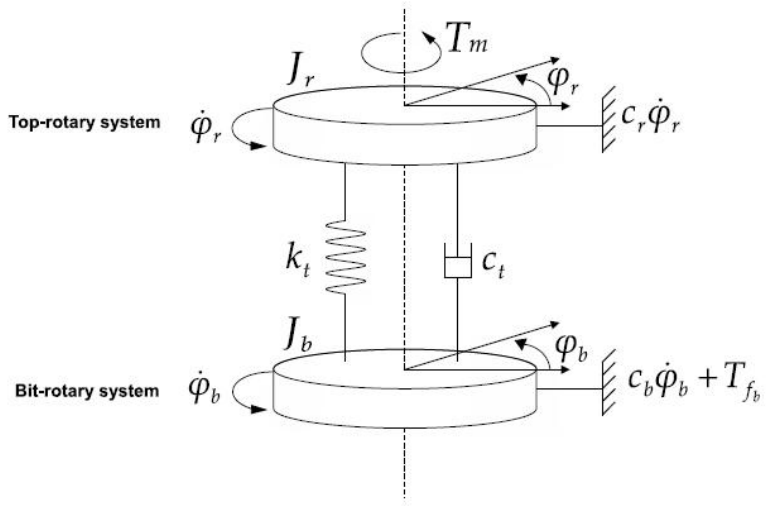

Figure 2.1: 2-DOF model describing the torsional behaviour of a simplified conventional vertical drillstring [19].

In order to accurately describe the model, multiple assumptions have been made about the system [19]:

1. The borehole and the drillstring are both vertical and straight.
2. No lateral bit motion is present.
3. The friction in the pipe connections, between the pipes and the borehole, are neglected.
4. The drilling mud is simplified by a viscous-type friction element at the bit.
5. The drilling mud fluids orbital motion is considered to be laminar, that is, without turbulence.
6. The Wob = Weight on the bit (not shown in Figure 2.1) is constant.

2.1 Equations of Motion

Under the assumptions given above and according to Figure 2.1, the equations of motion for this system are:

$$\begin{aligned} J_r\ddot{\varphi}_r + c_t(\dot{\varphi}_r - \dot{\varphi}_b) + k_t(\varphi_r - \varphi_b) &= T_m - T_r(\dot{\varphi}_r) \\ J_b\ddot{\varphi}_b - c_t(\dot{\varphi}_r - \dot{\varphi}_b) - k_t(\varphi_r - \varphi_b) &= -T_b(\dot{\varphi}_b), \end{aligned} \tag{2.1}$$

where $\varphi_i, \dot{\varphi}_i (i \in \{r, b\})$ are the angular displacements and angular velocities of the drillstring elements respectively, 'r' for rotary, and 'b' for bit.

We now look more closely at the equations of motion 2.1, this is a crucial step towards successfully simulating the system.

First, the top rotary system, J_r, has φ_r as the angle of rotation from a fixed axis, and consequently the rotary velocity is denoted $\dot{\varphi}_r$. T_m is the torque applied at the top rotary system, applied by an electrical motor at the surface of the drillstring. In this report T_m remains constant, with $T_m = u$, where u is one of our two control inputs. Also considered in the top rotary system equation is a viscous damping torque, this is modelled as some constant c_r multiplied by the velocity of the top rotary system, $\dot{\varphi}_r$, i.e.

$$T_r(\dot{\varphi}_r) = c_r\dot{\varphi}_r.$$

Similarly φ_b is the angle of rotation of the bit-rotary system with $\dot{\varphi}_b$ denoting the velocity of the bit. However, the torque on the bit is described as:

$$T_b(\dot{\varphi}_b) = c_b\dot{\varphi}_b + T_{f_b}(\dot{\varphi}_b),$$

with $c_b\dot{\varphi}_b$ approximating the influence of the mud drilling on the bit behavior, similar to that at the rotary system. But also included at the bit is $T_{f_b}(\dot{\varphi}_b)$. This is the friction torque modelling the bit-to-rock contact. There are multiple ways of modelling this friction, but for now the friction torque can be thought of as:

$$T_{f_b}(\dot{\varphi}_b) = W_{ob}R_b\left[\mu_{c_b} + (\mu_{s_b} - \mu_{c_b})\exp^{-\gamma_b|\dot{\varphi}_b|/\nu_f}\right]\operatorname{sgn}(\dot{\varphi}_b), \tag{2.2}$$

where weight on the bit, $W_{ob} > 0$, is our second control input. Also, $R_b > 0$ is the radius of the bit, $0 < \gamma_b < 1$ and $\nu_f > 0$, along with μ_{s_b}, $\mu_{c_b} \in (0, 1)$ which are the static and Coulomb friction coefficients associated with the drill bit inertia J_b [19].

Note that the static and Coulomb friction torques have the attached subscript 'b'. This is because we are interested in the friction torque at the bit rotary system. In this report we are not including a second friction torque, T_{f_r}, at the rotary system, as was proposed in [25].

In addition, the static and Coloumb friction torques can be defined as $T_{s_b} = W_{ob} R_b \mu_{s_b}$ and $T_{c_b} = W_{ob} R_b \mu_{c_b}$ respectively. This allows the friction torque formulae 2.2 to be re-written as:

$$T_{f_b}(\dot{\varphi}_b) = (T_{c_b} + (T_{s_b} - T_{c_b})\exp^{-\gamma_b |\dot{\varphi}_b|/\nu_f})\mathrm{sgn}(\dot{\varphi}_b). \tag{2.3}$$

The friction torque formulae 2.2 and 2.3 both make use of the 'sgn' function [19]. This is described as:

$$\begin{aligned} &\mathrm{sgn}(\dot{\varphi}_b) = \dot{\varphi}_b/|\dot{\varphi}_b|, \qquad \text{if } \dot{\varphi}_b \neq 0, \\ &\mathrm{sgn}(\dot{\varphi}_b) \in [-1, 1], \qquad \text{if } \dot{\varphi}_b = 0. \end{aligned} \tag{2.4}$$

From this definition it is clear that when the bit rotary velocity $\dot{\varphi}_b$, is non zero the sgn function takes the value $+1$ or -1, depending on the sign of $\dot{\varphi}_b$. But when $\dot{\varphi}_b = 0$, the bit rotary system has zero velocity and $\mathrm{sgn}(0) \in (0, 1)$. We therefore must use Utkin's Equivalent Control Method (explained previously in Section 1.3) to derive a suitable form for the friction torque at this point.

By defining the system state vector as $\boldsymbol{x} = (\dot{\varphi}_r, \varphi_r - \varphi_b, \dot{\varphi}_b)^T$, which corresponds to:

$$\begin{pmatrix} \text{The top rotary velocity} \\ \text{The difference in angle between the two inertia} \\ \text{The bit rotary velocity} \end{pmatrix}.$$

The equations of motion can be re-written as piecewise-smooth equations which fully describe the behaviour of the system [25]:

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \frac{1}{J_r}[-(c_t + c_r)x_1 - k_t x_2 + c_t x_3 + u] \\ x_1 - x_3 \\ \frac{1}{J_b}[c_t x_1 + k_t x_2 - (c_t + c_b)x_3 - T_{f_b}(x_3)] \end{pmatrix} \tag{2.5}$$

The state space representation of the system can be written as:

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \frac{-c_t+c_r}{J_r} & -\frac{k_t}{J_r} & \frac{c_t}{J_r} \\ 1 & 0 & 1 \\ \frac{c_t}{J_b} & \frac{k_t}{J_b} & -\frac{c_t+c_b}{J_b} \end{pmatrix} \boldsymbol{x} + \begin{pmatrix} \frac{u}{J_r} \\ 0 \\ -\frac{T_{fb}(x_3)}{J_b} \end{pmatrix}.$$

We next move onto understanding the theory behind the multiple types of behaviour the system can exhibit.

2.2 Theoretical Behaviour

There are three different types of long term behaviour patterns exhibited by this system. They arise due to the discontinuous surface in the friction torque when the bit rotary velocity is at zero, i.e. $\dot{\varphi}_b = x_3 = 0$. This system and it's behaviours have been investigated comprehensively by Dr Eva. M. Navarro-López in [19, 20, 21, 22, 23, 24, 25].

As mentioned previously, friction is a common cause of discontinuous systems because the friction acting on a body trying to start moving is greater in magnitude than the friction acting on the body when it is already moving. In our system this is modelled by the static and Coulomb friction torque, T_{sb} and T_{cb} respectively, with $T_{sb} > T_{cb}$. The bit rotary system must be given enough torque to overpower the static friction, T_{sb}, to start moving, then

the torque needed to continue moving decays to T_{cb}. This can be seen in the friction torque formulae, Equation 2.3, which uses exponential decay to model the reduction in friction:

$$T_{f_b}(\dot{\varphi}_b) = (T_{c_b} + (T_{s_b} - T_{c_b})\text{exp}^{-\gamma_b|\dot{\varphi}_b|/\nu_f}),$$

and it is shown in Figure 2.2. The graph clearly shows that as the bit rotary velocity increases, i.e. $|\dot{\varphi}_b| \to \infty$, the exponential term, $\text{exp}^{-\gamma_b|\dot{\varphi}_b|/\nu_f}$, decreases ($\to 0$), causing the friction torque $T_{f_b} \to T_{c_b}$.

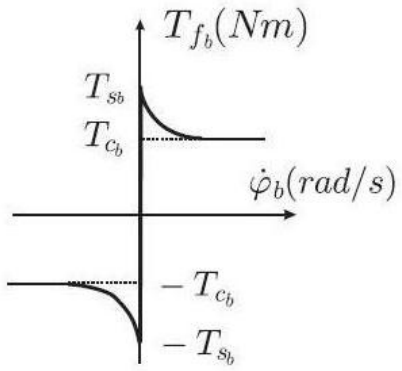

Figure 2.2: Friction at the bit rotary system, (T_{f_b}): $\dot{\varphi}_b$ (rad/s) is the bit angular velocity, $T_{s_b} = W_{ob}R_b\mu_{s_b}$, and $T_{c_b} = W_{ob}R_b\mu_{c_b}$ (Nm) are the static and Coloumb friction torques respectively [19].

When the bit rotary velocity is zero the friction it experiences could be anything ranging from T_{sb} to $-T_{sb}$, there is no way of knowing. Therefore $\dot{\varphi}_b = 0$ is known as the surface of discontinuity and is the key feature of the friction torque that produces strange behaviour when the velocity of the bit approaches it. This leads to three types of long term behaviour, these are:

1. Convergence to an equilibrium point with positive angular velocity.

2. Permanent stuck bit behaviour.

3. Stick-slip motion.

Convergence to an equilibrium point is the desired behaviour of the system. It is the case where both top and bit rotary systems continue turning with positive velocity, however $x_1 \neq x_3$. This means the drill is constantly twisted and $x_2 \neq 0$. This behaviour is safe and allows drilling to take place.

Permanent stuck bit behaviour is obtained when the bit rotary system has zero velocity, $x_3 = 0$. The bit may have rotated or oscillated many times, but eventually the rotary bit velocity remains zero for all further time. This is an unwanted outcome because when the bit inertia sticks, the electric motor at the top rotary system continues to turn the top inertia. This can be both dangerous and can damage the drillstring. The top rotary system eventually becomes stuck (under this behaviour), and often leads to abandoning expensive drillstrings.

Stick-slip motion is also unwanted and very harmful. It is the behaviour where the bit rotary velocity increases, then decreases (due to friction), then becomes zero. These three motions can repeat over indefinitely, with each not necessarily taking the same amount of time. The top rotary speed remains constant but when the velocity of the bit becomes zero, the turning force builds up (and under this behaviour) eventually overcomes the static friction with more torque than is needed to stay in motion. This results in speeds at the bit reaching up to six times the rotary speed measured at the surface due to the extra turning force. This is why the term stick-slip is used, as the bit sticks for a period of time before slipping. This behaviour is known as exhibiting sliding motion. Called so because when the bit is stuck for a period of time it is sliding within the surface of discontinuity, $x_3 = 0$.

We simulate each of the three behaviours in the next section. Learning more about how to model and simulate the surface of discontinuity of our drillstring system.

2.3 Simulations

To correspond to a real life drillstring, all simulations in this section use the following data which was extracted from [1]:

$$J_r = 2122kgm^2, J_b = 471.9698kgm^2, R_b = 0.155575m,$$

$$k_t = 698.063Nm/rad, c_t = 139.6126Nms/rad, c_r = 425Nms/rad,$$

$$c_b = 50Nms/rad, \mu_{c_b} = 0.5, \mu_{s_b} = 0.8, D_\nu = 10^{-6}, \gamma_b - 0.9, \nu_f = 1.$$

To adequately model the discontinuous plane at $x_3 = 0$, the original friction torque has been replaced by a simulation torque [22]:

$$T_{f_b}(\boldsymbol{x}) = \begin{cases} T_{e_b}(\boldsymbol{x}) & \text{if } |x_3| \leq \delta, |T_{e_b}| \leq T_{s_b} \text{ (Stuck)}, \\ T_{s_b}\text{sgn}(T_{e_b}(\boldsymbol{x})) & \text{if } |x_3| \leq \delta, |T_{e_b}| > T_{s_b} \text{ (Stick-to-slip transition)}, \\ f_b(x_3)\text{sgn}(x_3) & \text{if } |x_3| > \delta, \text{ (Sliding)}, \end{cases} \tag{2.6}$$

where the friction torque $f_b(x_3) = T_{c_b} + (T_{s_b} - T_{c_b})\text{exp}^{-\gamma_b|\dot{\varphi}_b|/\nu_f}$. The reaction torque $T_{e_b} = c_t(x_1 - x_3) + k_t x_2 - c_b x_3$, plays the roll of the equivalent control, U_{eq}, from Utkin's Equivalent Control Method in order to obtain the dynamics on the surface of discontinuity $x_3 = 0$.

This model introduces a zero band of size $Dv = \delta$, whereby if the velocity is less than this value, the system can be either stuck or in transition. This allows simulation software such as MatLab to easily detect when the velocity is close to the surface of discontinuity. This avoids the problems we encountered with our naively programmed bouncing ball, namely those associated with attempting to integrate too close to the surface of discontinuity. This friction torque model is a combination of a switch model [17] and the Karnopp model [15], more will follow in Section 2.4 about the use of different friction models. But for now, we are ready to simulate each of our three behaviours using MatLab.

2.3.1 Convergence to Equilibrium

To simulate behaviours in MatLab we use the code presented by Rebekah Carter in [10], which can be found in Appendix C. Each of the three different long term behaviours is attainable inside a timespan of [0, 100] by altering the control parameters u and W_{ob} intelligently. In each of the three example cases shown here the motor torque $T_m = u$ has been kept constant at $4300Nm$ and we have altered the weight on the bit to greatly effect the behaviour.

Figure 2.3 shows the convergence to equilibrium behaviour. The left plot shows the rotary velocities of the top and bit inertias, red and blue respectively. The right plot shows the state space trajectory in 3D, with axis $(\dot{\varphi}_r, \dot{\varphi}_r - \dot{\varphi}_b, \dot{\varphi}_b) = (x_1, x_2, x_3)$.

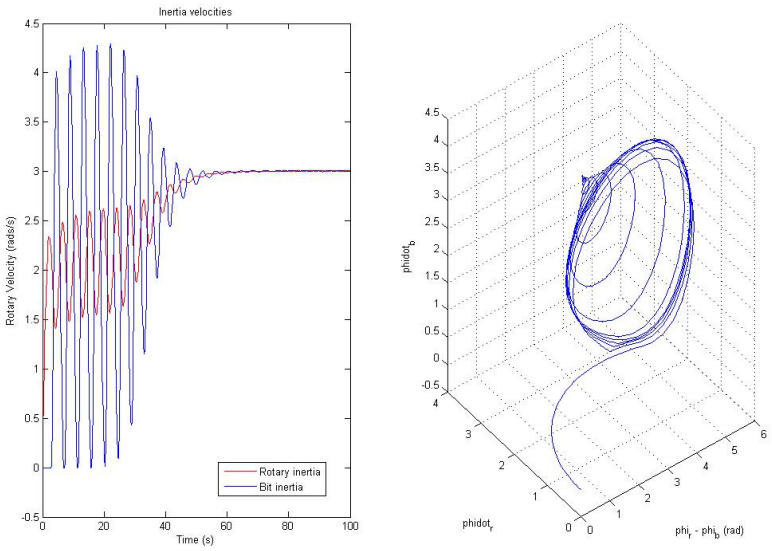

Figure 2.3: Rotary velocities of top and bit inertia (left) and state space trajectories (right) of sliding motion using $u = 4300Nm$ and $W_{ob} = 35,500N$.

The left plot clearly shows that eventually, both rotary system velocities tend

to some positive constant. Another feature of the graph is that it shows the top rotary system (red) starts turning before the bit rotary system (blue). The bit starts rotating some time after, due to the motor turning force being applied at the surface. The two inertia also maintain $x_1 - x_3 \neq 0$ when $x_1, x_2 > 0$. The right plot clearly shows the trajectory's transient before the equilibrium is reached.

2.3.2 Permanent Stuck Bit

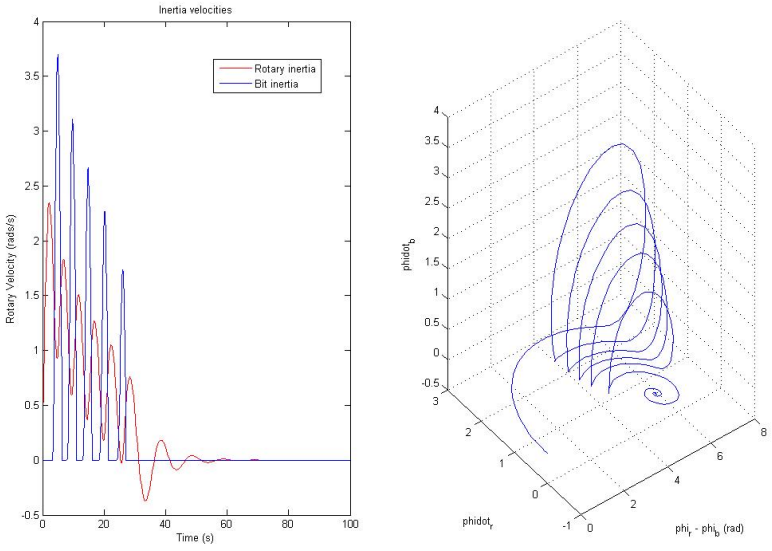

Figure 2.4: Rotary velocities of top and bit inertia (left) and state space trajectories (right) of permanently stuck bit behaviour using $u = 4300Nm$ and $W_{ob} = 41,000N$.

Figure 2.4 is simulated using the same MatLab code but by altering the W_{ob} to 41,000N. There becomes too much friction present at the bit rotary system for the electric motor situated at the top inertia to overcome. When the bit becomes stuck the top inertia remains turning trying to overcome the even larger static friction torque. If it cannot, it too becomes stuck. In

this example, just before the top inertia is stuck indefinitely, it releases it's energy by rotating with a negative velocity. This means the drillstring turns in reverse for a period of time, further adding to why this type of behaviour is dangerous and unwanted.

2.3.3 Stick-slip Motion

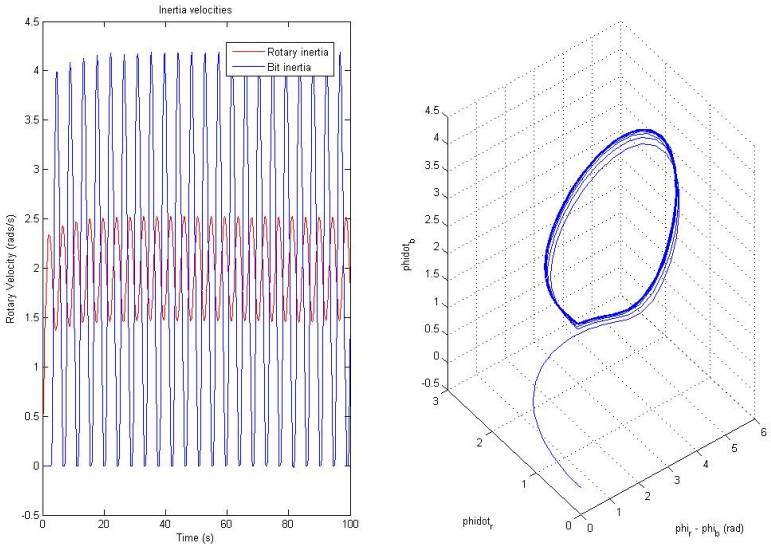

Figure 2.5: Rotary velocities of top and bit inertia (left) and state space trajectories (right) of stick-slip motion using $u = 4300Nm$ and $W_{ob} = 36,000N$.

By reducing the W_{ob} slightly to 36,000N the top rotary system can overpower the static friction torque when the bit has zero rotary velocity. In this case this leads to the never ending pattern of behaviour known as stick-slip motion. Whilst trying to overpower the friction of the stuck bit, the top inertia begins to slows down. After the bit is finally released it's rotary velocity shoots up, far exceeding that of the top inertia. As the top inertia starts to speed up once more, the bit slows to a maintainable velocity but gets stuck once more due to the larger friction at slower velocity. This behaviour

is shown in Figure 2.5 (left). The trajectory plot (right) clearly shows this behaviour as oscillatory with a transient followed by the periodic orbit.

After simulating these behaviours it is worth noting that we obtained all three types without changing one of our control parameters, u. This leads us to the conclusion that for each value of u there are different ranges of the parameter W_{ob} which will produce the three behaviours. Even more importantly is that these ranges are ordered: the W_{ob} needed to produce the convergence to equilibrium behaviour is less than that needed to produce stick-slip behaviour, which in turn is less than the W_{ob} needed to produce stuck behaviour. This is all due to the W_{ob} term being crucial to the friction torque at the bit. This conclusion is also something discovered by R. Carter when she investigated the long term behaviours of the parameter values in [10].

2.4 Friction Models

There are many ways to model the bit-to-rock type friction present in our system, $T_{f_b}(x_3)$. We have already seen one friction torque, Equation 2.6. We proceed by looking at other possible methods to model the friction torque, $f_b(x_3)$. Looking at the advantages and possible limitations to each.

By anticipating problems with the friction torque model, namely due to the exponential function, we will be prepared if we have to alter it when we advance our work with verification software. Next, we detail some similar models which could be used instead in the event we encounter such problems.

The original friction torque, Equation 2.6, can now be re-introduced as a 'Karnopp's friction model with a decaying exponential-type friction in the slipping phase' which was proposed in [25]. This model is mapped over to our example system by ignoring the friction torque at the top inertia. After

applying such a change the friction model has the form:

$$T_{f_b}(\boldsymbol{x}) = \begin{cases} \min\{|T_{e_b}(\boldsymbol{x})|, T_{s_b}\} sgn(T_{e_b}(\boldsymbol{x})) & \text{if } |x_3| < Dv \\ f_b(x_3) sgn(x_3) & \text{if } |x_3| \geq Dv \end{cases} \tag{2.7}$$

with,

$$\begin{aligned} T_{e_b}(\boldsymbol{x}) &= c_t(x_1 - x_3) + k_t x_2 - c_b x_3, \\ f_b(x_3) &= W_{ob} R_b \mu_b(x_3), \\ \mu_b(x_3) &= \mu_{c_b} + (\mu_{s_b} - \mu_{c_b}) e^{-\gamma_b |x_3|}. \end{aligned} \tag{2.8}$$

2.4.1 Taylor Series Approximation

As mentioned, we anticipate problems in future work due to the exponential term in the friction torque model. We therefore replace it with the Taylor Series approximation:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \text{for all } x.$$

This does not effect the dynamics on the surface of discontinuity, so T_{e_b} remains the same. But the friction torque equation in 2.8 now has the form:

$$\begin{aligned} f_b(x_3) &= W_{ob} R_b \mu_b(x_3), \\ \mu_b(x_3) &= \mu_{c_b} + (\mu_{s_b} - \mu_{c_b}) \Big(1 + (-\gamma_b |x_3| / v_f) + (-\gamma_b |x_3| / v_f)^2 / 2! + ... \\ &\quad ... + (-\gamma_b |x_3| / v_f)^3 / 3! + (-\gamma_b |x_3| / v_f)^4 / 4! \Big) * sgn(x_3). \end{aligned} \tag{2.9}$$

We know, due to the approximation of the exponential function, this model will not produce exactly the same behaviour for all parameter values as the original. However, similarities between the two models obviously exist. This approximate friction model still makes use of a zero band to simulate the discontinuous system more easily. It uses the same T_{e_b} as the equivalent control, U_{eq}, to obtain dynamics on the surface of discontinuity, $x_3 = 0$, in exactly the same way as the original friction torque in Equation 2.6. For these reasons the friction torque at the bit appears the same and is appreciated in Figure

2.6 (left).

The Function Handle MatLab code for this alternate friction torque equation can be found in Appendix C.2. We simulate the system with $u = 3500Nm$ and W_{ob} varying from $29000N$ to $35000N$ over a timespan of [0,100]. An initial comparison of results is given in Table 2.1.

Since seven of the nine simulations (78%) produced the same long term behaviour as the original friction model, we accept this Taylor Series approximation as a possible replacement model. We will however, only need this replacement model if we incur significant problems with the exponential term in future work.

Comparison of behaviours using different friction models			
u(Nm)	W_{ob}(N)	Behaviour Model 2.6	Behaviour Model 2.7-2.9
3500	29000	Positive vel.	Positive vel.
3500	30000	Positive vel.	Stick-slip
3500	30050	Positive vel.	Stick-slip
3500	30100	Stick-slip	Stick-slip
3500	30250	Stick-slip	Stick-slip
3500	30500	Stick-slip	Stick-slip
3500	31000	Stick-slip	Stick-slip
3500	32500	Perm. Stuck	Perm. Stuck
3500	35000	Perm. Stuck	Perm. Stuck

Table 2.1: Comparison between the original friction torque, Equation 2.6 and the Taylor Series approximation to the exponential, Equation 2.9.

2.4.2 Quotient-type Friction Torque

We now move on to look at a second variant of the friction torque equation. This will give us another option if we incur any problems with the original.

We are not however changing the friction torque model drastically. This method still makes use of a Karnopp's friction model, Equation 2.7, but allows the friction torque in the slipping phase to be ruled by a quotient-type decaying law [25]:

$$\begin{aligned} f_b(\dot{\varphi}_b) &= W_{ob} R_b \mu_b(\dot{\varphi}_b), \\ \mu_b(\dot{\varphi}_b) &= \mu_{s_b} + \Big(\mu_{s_b} - \mu_{c_b}/(1+\gamma_b|\dot{\varphi}_b|)\Big). \end{aligned} \tag{2.10}$$

The friction torque is again represented in Figure 2.6 (left). The MatLab code for this friction model is attached in Appendix C.4. We do not simulate the system here however. We present this friction torque only to provide an alternative model for when we use verification techniques later in the project.

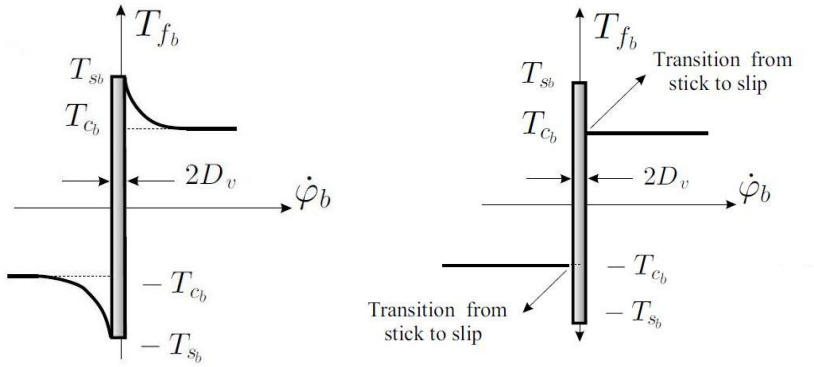

Figure 2.6: Representation of exponential, approximation to exponential and quotient-type decaying friction torque, Equations 2.8, 2.9 and 2.10 (left). Non-exponential friction torque representation, Equation 2.11 (right) [25].

2.4.3 Combination of Dry Friction, Switch and Karnopp's Models

To get away from this type of decreasing friction torque, a combination of models can be used. Proposed in [16] is a model which incorporates a switch

model along with a dry friction and Karnopp's models. Ignoring the friction torque at the top inertia in the same way as before, we obtain the model:

$$T_{f_b}(\boldsymbol{x}) = \begin{cases} T_{e_b}(\boldsymbol{x}) & \text{if } |x_3| < Dv, |T_{e_b}| \leq T_{s_b} \text{ (Stuck)}, \\ T_{s_b}\text{sgn}(T_{e_b}(\boldsymbol{x})) & \text{if } |x_3| < Dv, |T_{e_b}| > T_{s_b} \text{ (Stick-to-slip transition)}, \\ T_{c_b}\text{sgn}(x_3) & \text{if } |x_3| \geq \delta, \text{ (Sliding)}, \end{cases} \tag{2.11}$$

Friction torque Equation 2.11 obviously differs in the sliding phase from the previous models. It was proposed in [16] alongside a set of parameters for a prototype example of the drillstring system [18]. They do not correspond to a real drillstring and so straight comparisons between this and our simulations in Section 2.3 is meaningless. They can however still be used to model each of the long term behaviours and to understand the slipping phase of the system better.

Friction torque Equation 2.11 is shown in Figure 2.6 (right). It is obviously more simple than the models which express some form of decaying friction torque in the slipping phase. Because of this, it is likely to be simpler to simulate and to use in future verification work. We have attached the MatLab Driverscript and Function Handle in Appendix C.5 and C.6 respectively.

The multiple friction models looked at in this section will give us options once we have passed the discontinuous system into verification software.

2.5 Hybrid Automata Models

We now introduce a hybrid automaton approach to represent the example system. We look at two hybrid automata models which are based on our discontinuous example: first a 3-location model and second a 5-location model, both taken from [23]. We will then continue and present both models in the verification software KeYmaera in Chapter 4.

2.5.1 Three Location Representation

We start with the 3 location automaton, $Q = \{q_1, q_2, q_3\}$. It is clear that one of the locations of this model must be the stuck location, $x_3 = 0$. The other two are the positive and negative velocity slip locations $x_3 > 0$ and $x_3 < 0$ respectively. This three location model makes use of the original friction torque model from Equation 2.3. Along with Utkin's Equivalent Control Method in order to find the dynamics on the surface of discontinuity, which in Section 2.3 was found to equal the reaction torque, T_{e_b}. Figure 2.7 shows the directed graph representing this automata with $\{q_1, q_2, q_3\} = \{slip^+, slip^-, stuck\}$. We proceed by understanding the multiple domains, guards and resets that it includes.

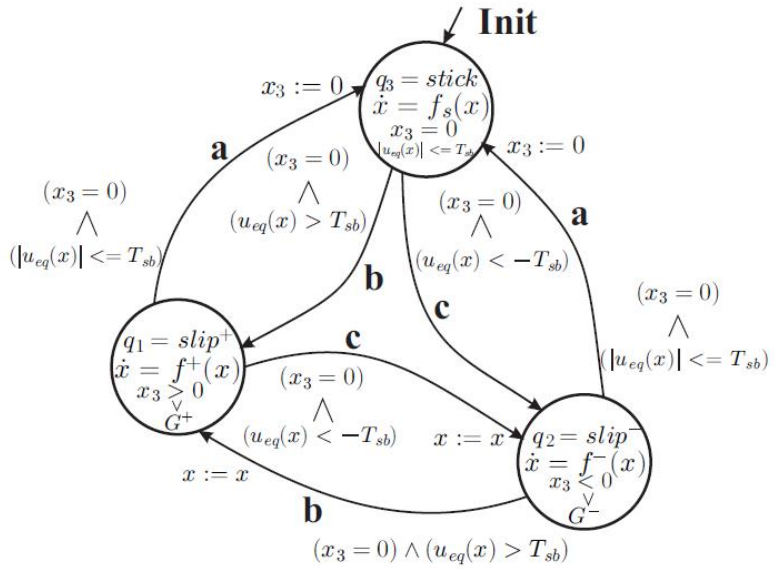

Figure 2.7: Directed graph associated with the three location automaton of the drillstring system [23].

Stuck location

The stuck location, q_3, is crucial to the system. When the trajectory has bit rotary velocity zero, $x_3 = 0$, it is in this location. This occurs due to either being initialized in this location as well as if the bit inertia becomes stuck, due to the friction torque, after some transient behaviour. We define the region of zero bit velocity as the switching surface: $S_0^b = \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 = 0\}$.

The domain specified for this location in Figure 2.7 also has an additional condition:

$$|U_{eq}(\boldsymbol{x})| \leq T_{s_b}.$$

That is, the system is in the stuck location, only when the trajectory is in the switching surface and the absolute value of the equivalent control is less than the static friction torque. So the domain of this location becomes:

$$Dom(stuck) = S_s^b = \{\boldsymbol{x} \in S_0^b : |U_{eq}(\boldsymbol{x})| \leq T_{s_b}\},$$

which is known as the sliding set. This domain makes sense because if the equivalent control, which specifies the dynamics on the surface of discontinuity $x_3 = 0$, is greater than the static friction torque, the bit inertia would begin to move and the velocity would increase. This is also the reason the guard condition on the transitions out of the stuck location to the positive and negative slip locations, (q_3, q_1) and (q_3, q_2), are:

$$G(q_3, q_1) = \{\boldsymbol{x} \in S_0^b : U_{eq}(\boldsymbol{x}) > T_{s_b}\},$$
$$G(q_3, q_2) = \{\boldsymbol{x} \in S_0^b : U_{eq}(\boldsymbol{x}) < -T_{s_b}\}.$$

When the trajectory doesn't satisfy either of these guard conditions however, it remains in this location and evolves under the dynamics:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}_s(\boldsymbol{x}, u) = \begin{pmatrix} \frac{1}{J_r}[-(c_t + c_r)x_1 - k_t x_2 + u] \\ x_1 \\ 0 \end{pmatrix}. \tag{2.12}$$

Slip$^+$ and Slip$^-$ locations

The two slip locations represent the sliding motion the system can exhibit. This occurs either when the bit velocity is non-zero, $x_3 \neq 0$, or when the trajectory is in the switching surface and the equivalent control is greater in magnitude than the static friction torque, $|U_{eq}(\boldsymbol{x})| > T_{s_b}$. The domains for these two locations are described as:

$$\begin{aligned} Dom(slip^+) &= \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 > 0\} \cup G^+, \\ Dom(slip^-) &= \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 < 0\} \cup G^-, \end{aligned} \tag{2.13}$$

where,

$$\begin{aligned} G^+ &= \{\boldsymbol{x} \in S_0^b : U_{eq} > T_{s_b}\}, \\ G^- &= \{\boldsymbol{x} \in S_0^b : U_{eq} < -T_{s_b}\}. \end{aligned} \tag{2.14}$$

The dynamics of the trajectory in these locations is detailed above in Equation 2.5. Where the sign of the velocity at the bit refers to which side of the surface of discontinuity the trajectory is on:

$$\dot{\boldsymbol{x}} = \begin{cases} f^+(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{X}^+ \\ f^-(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{X}^-. \end{cases}$$

There are two different transitions out of the sliding locations: to the stuck location or to the other sliding location. The guard condition from a slip location to the stuck location is easy to understand. It is simply the opposite condition to leaving the stuck location: the trajectory is in the switching surface and the equivalent control is not greater in magnitude than the static friction torque:

$$G(q_1, q_3) = G(q_2, q_3) = \{\boldsymbol{x} \in S_0^b : |U_{eq}(\boldsymbol{x})| \leq T_{s_b}\}.$$

Secondly, there are the transitions which pass straight through the switching surface, i.e. from location q_1 to q_2 or the reverse. These have the guard condition:

$$\begin{aligned} G(q_1, q_2) &= \{\boldsymbol{x} \in S_0^b : U_{eq}(\boldsymbol{x}) < -T_{s_b}\}, \\ G(q_2, q_1) &= \{\boldsymbol{x} \in S_0^b : U_{eq}(\boldsymbol{x}) > T_{s_b}\}. \end{aligned}$$

The reset conditions in this automata are simple also. On entering the stuck location from the slip locations, x_3 is reset to zero. On entering the slip regions, the reset $x := x$ simply keeps the value of state vector $\boldsymbol{x}$ unchanged. Note ':=' means assignment.

Each of the three locations have been explained, along with their dynamics and transitions. This is an important automata which we will revisit later with verification software.

2.5.2 Five Location Representation

We look also at a five-location automaton. This makes use of the friction torque model used to simulate in Section 2.3, detailed above in Equation 2.6.

It has two extra locations because of the stick-to-slip transitions present in this friction model. This model also makes use of a zero band around the surface of discontinuity, to make it is easier to simulate. We therefore have five locations: $Q = \{q_1, q_2, q_3, q_4, q_5\} = \{slip^+, slip^-, stuck, trans^+, trans^-\}$. We again use Utkin's Equivalent Control Method to describe the dynamics on the surface of discontinuity, $U_{eq}(\boldsymbol{x}) = T_{e_b}$. Figure 2.8 shows the directed graph representing this automata.

This is an important automata. We aim to pass the drillstring system into KeYmaera using this representation also. Because we revisit this system later, to verify its properties, it is prudent that we detail the five domains of

the automata model here:

$$
\begin{aligned}
&Dom(slip^{+}) = [x_3 > \delta] && \text{with } T_{f_b} = f_b(x_3),\\
&Dom(slip^{-}) = [x_3 < \delta] && \text{with } T_{f_b}(\boldsymbol{x}) = -f_b(x_3),\\
&Dom(stuck) = [x_3 \leq \delta] \text{ and } [|U_{eq}(\boldsymbol{x})| \leq T_{s_b}] && \text{with } T_{f_b}(\boldsymbol{x}) = T_{e_b},\\
&Dom(trans^{+}) = [x_3 \leq \delta], [U_{eq}(\boldsymbol{x}) > T_{s_b}] && \text{with } T_{f_b}(\boldsymbol{x}) = T_{s_b},\\
&Dom(trans^{-}) = [x_3 \leq \delta], [U_{eq}(\boldsymbol{x}) < -T_{s_b}] && \text{with } T_{f_b}(\boldsymbol{x}) = -T_{s_b}.
\end{aligned}
\tag{2.15}
$$

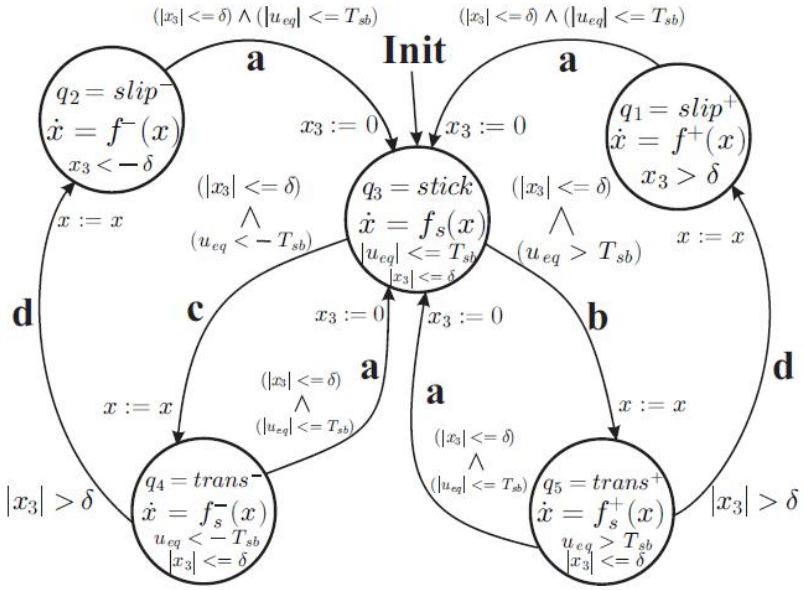

Figure 2.8: Directed graph associated with the five location automaton of the drillstring system [23]

Stuck location

With the exception of the zero band technique, which introduces a small region of size δ around $x_3 = 0$, the domain for the stuck location remains the same as in the three state automaton representation. The dynamics in this location also remain the same as Equation 2.12. However, when leaving

this location the trajectory now moves to the transition locations instead of straight to the slip locations. These transitions have guard conditions:

$$G(q_3, q_4) = \{|x_3| \leq \delta \wedge U_{eq}(\boldsymbol{x}) < -T_{s_b}\},$$
$$G(q_3, q_5) = \{|x_3| \leq \delta \wedge U_{eq}(\boldsymbol{x}) > T_{s_b}\}.$$

On the event of overcoming one of the guard conditions, a discrete transition takes place where the resets do not change the value of $\boldsymbol{x}$.

Trans$^+$ and Trans$^-$ locations

As mentioned above, two extra states are needed to handle the transitions from stick to slip using this friction torque model. These are locations from which only the same signed slip location and stuck locations are reachable. The guard conditions are:

$$G(q_4, q_2) = G(q_5, q_1) = \{|x_3| > \delta\},$$
$$G(q_4, q_3) = G(q_5, q_3) = \{|x_3| \leq \delta \wedge |U_{eq}(\boldsymbol{x})| \leq T_{s_b}\}.$$

The resets again set x_3 to zero on entry to the stuck location and leave $\boldsymbol{x}$ unchanged on entry to the slip locations.

Since x_3 is still less than some small value δ whilst in this location, the system continues to evolve with the dynamics defined in Equation 2.12. It is clear however that the trajectory is on one side of the surface of discontinuity, therefore:

$$\dot{\boldsymbol{x}} = \begin{cases} f_s^+(\boldsymbol{x}) & \text{if } U_{eq}(\boldsymbol{x}) > T_{s_b}, \\ f_s^-(\boldsymbol{x}) & \text{if } U_{eq}(\boldsymbol{x}) < -T_{s_b}. \end{cases}$$

Slip$^+$ and Slip$^-$

Due to the intermediate transition locations when moving from the stuck to the slip locations, the domain of both slip locations are more simple than in the three location automata representation. With the introduction of the

zero band, the domains are:

$$\begin{aligned} Dom(slip^{+}) &= \{x_3 > \delta\}, \\ Dom(slip^{-}) &= \{x_3 > -\delta\} \end{aligned} \tag{2.16}$$

A trajectory in either of these domains evolves again with the dynamics specified in Equation 2.5. The guard conditions to the only reachable state, the stuck location, are again similar to those in the three state automata. They are altered only due to the zero band introduction, they are:

$$G(q_1, q_3) = G(q_2, q_3) = \{|x_3| \leq \delta \wedge |U_{eq}(\boldsymbol{x})| \leq T_{s_b}\}.$$

Similarly, x_3 is reset to zero on making either of these discrete transitions.

We have given full details of two important hybrid automata representations of our discontinuous example system. Both we will revisit and use to pass the drillstring system into verification software.

Chapter 3

Formal Verification and KeYmaera tool

This report focuses on formally verifying behaviour of our discontinuous dynamical system, previously seen in Chapter 2. In this chapter we start by describing what formal verification is and how it can be used. We introduce a formal verification tool called KeYmaera. It is based on the KeY Project but has been especially designed for the verification of hybrid systems. We look at some techniques it uses to verify properties of such systems. In Chapter 4 we move on to passing the drillstring system into KeYmaera to verify safety conditions. This will allow us to obtain regions of control parameters for each of the long term behaviours. Finally we aim to use the results to reduce unwanted and catastrophic behaviours from the system.

3.1 Formal Verification

Formal verification is a technique where some property of a system can be checked in an automated way. As a technique, it arose to check correctness properties of computer programs. Recently however it has been mapped over to dynamical systems theory and is commonly used to verify various properties about systems in this field. Verification of dynamical systems is

important because once a system has had properties successfully verified, no strange results or unexpected behaviour can occur.

A lot of research is currently being undertaken into the verification of the safety property of hybrid systems. This property has become increasingly important to verify about hybrid systems due to their use in everyday, safety critical applications.

3.1.1 Safety Verification

A safety property is a property stating that nothing bad ever happens. This can be during the execution of a computer program or in a trajectory of a dynamical system. Verifying safety properties of hybrid dynamical systems amounts to finding an answer to the question: 'Is a potentially unsafe configuration, or state, reachable from an initial configuration?' (Extract from [33]). The theory behind this method of verifying a safety condition uses the concept of reachable sets.

Reachable Sets

This is a property where initial conditions of a system are specified, and the set of all states that are reachable from this initial configuration are found. This is called the forward reachable set.

For a backward reachable set, a final or target set of states is specified initially. Then the set of states from which trajectories start, that can reach the target set, are sought after. Figure 3.1 shows the difference between the two definitions. Note the backwards reachable set is not just a time reversed version of the forward reachable set.

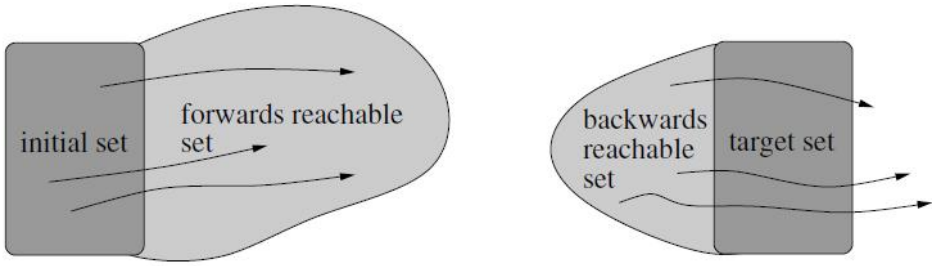

Figure 3.1: Difference between reachable sets: Forward (left) and Backwards (right) [33].

3.1.2 Additional Properties

Verification of dynamical systems can be handled in different ways often with the use of additional properties. One such additional property, the correctness property, is itself a safety property. It states that if a condition is initially true (pre-condition), then after all executions of a system, it satisfies a post-condition. The post-condition can then be claimed to hold for all reachable states of the dynamical system.

Safety properties are complemented by liveness properties. A liveness property is a property stating that something good will eventually happen. Again this has been mapped over to dynamical systems theory. It implies that a trajectory will eventually visit a set of states that are favourable.

In the next section we describe the software tool KeYmaera. This uses safety, liveness, reactivity and controllability properties to prove properties of hybrid systems.

3.2 Formal Verification Tool: KeYmaera

The software tool KeYmaera is based upon the formal software system KeY. This system uses a novel theorem prover on first-order dynamic logic for Java card programs. KeYmaera is a similar theorem prover but for differential dynamic logics. As the central concept, it implements an axiomatization of the transition behaviour of hybrid systems. It does this by introducing formal properties which need to hold under differential dynamic logic, d$\mathcal{L}$. If these properties hold in d$\mathcal{L}$, we are able to prove correct functioning hybrid systems. More on this logic will follow in Section 3.2.2.

KeYmaera can therefore check post-conditions of hybrid systems in an automated way. Using our drillstring system, we will alter the pre-conditions and obtain control parameters for which it satisfies a safety post-condition. First, we must understand KeYmaera and we begin this by looking at some of the key features of the software.

3.2.1 Implementation

KeYmaera has been created by implementing the deductive theorem prover, KeY, and the computer algebra system, Mathematica.

- KeY is an interactive theorem prover which has been generalized from working with discrete systems to hybrid systems by adding the support for differential dynamic logic, d$\mathcal{L}$. This logic was proposed by André Platzer in [27]. This was due to limitations in handling system parameters, when verifying hybrid systems, using verification methods such as model checkers.

- Mathematica or similarly Orbital, the math library for Java, is used to obtain symbolic solutions of differential equations. This is used to handle continuous dynamics of a system.

KeYmaera follows a deductive approach. This means that in a purely syntactic basis, it demonstrates that one formula is a logical consequence of another formula. Whilst following this approach it uses differential dynamic logic to aid its fully symbolic technique. We investigate this along with the program notation, d$\mathcal{L}$, in the next section. However, this report does not detail differential dynamic logic fully. For additional information refer to [26, 27].

3.2.2 Differential Dynamic Logic

Introduced in [26, 27] is differential dynamic logic, d$\mathcal{L}$. Along with its verification logic, it forms the basis of the software tool KeYmaera. It is essentially program notation for hybrid systems which acts as a foundation for deductive verification of such systems. The logic supports programs with discrete and differential actions making it very useful.

The program notation has been specifically designed to have compositional semantics. KeYmaera uses a technique whereby properties of hybrid programs are reduced to properties of their parts. Therefore the compositional semantics are exploited to verify correctness of hybrid systems. The technique is known as symbolic decomposition. More will follow on this technique in Section 3.2.3. First we must understand how systems are passed into hybrid program language.

Hybrid automata are represented by discrete transitions and continuous dynamics with non-deterministic choice ($\cup$) between the transitions. This allows KeYmaera to express them using a sequence of statements in hybrid program language. All hybrid programs are built with the statements in Table 3.1.

The foundation of the specification and verification logic of KeYmaera is first-order logic over the reals. In Table 3.1, F represents a formula of first-order real arithmetic and α, β represent hybrid programs. It is important to

note that all hybrid automata can be represented as hybrid programs.

Statement	Effect
$\alpha;\beta$	Sequential composition, first performs α and then β.
$\alpha \cup \beta$	Nondeterministic choice, following either α or β.
α^*	Nondeterministic repetition, repeating α $n \geq 0$ times.
$x := \theta$	Discrete assignment of the value of term θ to variable x (jump).
$x := *$	Nondeterministic assignment of an arbitrary real number to x.
$(x_1' = \theta_1, \ldots,$ $x_n' = \theta_n, F)$	Continuous evolution of x_i along differential equation system $x_i' = \theta_i$, restricted to a maximum domain or invariant region F.
?F	Check if formula F holds at current state, abort otherwise.
α ++ β	Non-deterministic choice following either α or β.
if (F) then α	Perform α if F holds, do nothing otherwise.
if (F) then α else β	Perform α if F holds, perform β otherwise.

Table 3.1: Syntax statements of hybrid programs within KeYmaera [30].

For expressing correctness statements about hybrid systems however, this table is complemented by modal operators. Denoted $[\alpha]$ and $< \alpha >$, named box-operator and diamond-operator respectively, for a hybrid program α. Modal operators refer to states reachable by hybrid program α and can be placed in front of any formula.

With this we can start to see how hybrid systems are passed into KeYmaera. We are now able to construct formulae of the form:

$$[\alpha]\phi.$$

This is a safety condition and if it is verified as being true, it means all states reachable by hybrid program α, satisfy the formula ϕ (post-condition). If we use a diamond box-operator: $< \alpha > \phi$. The formula now expresses a liveness property, that is, by following hybrid program α, there is at least one state reachable in which formula ϕ holds.

A step further is to define Hoare-triples. A Hoare triple $\{\psi\}\alpha\{\phi\}$ can be expressed in KeYmaera by the formula:

$$\psi \rightarrow [\alpha]\phi.$$

This formula is true if, and only if, all states reachable by hybrid program α satisfy post-condition ϕ, when starting from an initial state that satisfies ψ.

Generally speaking, this is what we need to do with our drillstring system. We specify initial conditions, pass the discrete transitions and continuous dynamics of the system into KeYmaera as a hybrid program and enter a post-condition. This method will enable us to test safety regions of control parameters by using a safe domain as the post-condition.

Before we continue and pass our drillstring system into program notation, we take a closer look at some of the techniques used by KeYmaera to prove properties about hybrid programs.

3.2.3 Symbolic Decomposition

As mentioned above, hybrid programs have perfectly compositional semantics. This is due to the proposed dynamic logic being closed under logical connectives, allowing hybrid programs to be syntactically decomposed into fragments within KeYmaera: these subprograms correspond to partial executions of part of the full program. The semantics of a compound hybrid program therefore, is a simple function of the semantics of its fragments.
i.e. KeYmaera can express simultaneous correctness statements about multiple fragments of a hybrid program α, using conjuncts $[\alpha_1]\phi_1 \wedge [\alpha_2]\phi_2$.

This is an important feature, as the verification algorithm used in KeYmaera recursively decomposes a given hybrid program into fragments $\alpha_1, \ldots, \alpha_n$ (e.g. to find local invariants for each α_i) and recombines corresponding cor-

rectness statements about these fragments later. Further detail on this is given in [28].

3.2.4 Symbolic Execution

This is a technique that can be used to check the correct behaviour of given hybrid programs. It is used in KeYmaera for verification purposes and for static analysis of programs. It is a method whereby a program is executed without concrete values of any of the variables occurring in the program. By doing this we get different paths through the program, split due to 'if ... then' or 'if ... then ... else' statements in the program.

After symbolic execution has taken place, there is a defined number of paths or possible routes through a program. Given a set of pre-conditions, we know which path is chosen and what the final value of the variables will be. We use a short example to show how this works.

Example 1 (Extracted from [6]).
This program sets the value of the variable, x, to 0 if it is less than or equal to it, and it decrements x by 1 if it is greater than 0:

```
1     {if (x > 0) then
2        x = x - 1
3      else
4        x = 0;
5      return x; }
```

We say that x is an arbitrary integer when the program starts. If the condition:

```
if (x > 0),
```

holds, our single original path splits into two paths due to the 'if ... then ... else' statement. In the first path we know that x is greater than 0, and

in the second we know that x is less than or equal to 0. We continue and assign the new values of x:

$$\begin{aligned} \text{Path 1:} \quad & (x > 0) \quad \Rightarrow x = x - 1, \\ \text{Path 2:} \quad & x \leq 0 \quad \Rightarrow x = 0. \end{aligned}$$

This technique becomes more useful once we bind local variables to program variables with the use of 'updates'. An update contains the current value of all program variables, they are located in front of the program modality. To bind a logical variable to a program variable we just write an update of the form {Program Variable := LogicalVariable} [6]. Using updates, we can see a possible proof for the program in Example 1.

Example 2 (Extracted from [6])
Using the same program as in Example 1, we create a proof case of the form $\psi \rightarrow [\alpha]\phi$ by adding logical pre- and post-conditions, ψ and ϕ, and using the modal operators, [ ], around the program:

```
1  ∀ int x0
2  ⇒
3    {x := x0}
4       [{if (x > 0) then
5            x = x - 1
6       else
7            x = 0;
8       return x;
9       } ]
10   (x ≥ 0)
```

Because it is universally quantified, x_0 is now a logical variable. The update $\{x := x_0\}$ sets the value of the program variable x to the value x_0. The execution is then split and we have one update for each of the two paths present in the program. In Path 1 $(x > 0)$ we have the update $\{x := x - 1\}$

and in Path 2 ($x \leq 0$) we have $\{x := 0\}$. As we can see, the post-condition (or safety condition) holds for both paths. The reader is referred to [6] for more details on symbolic execution.

3.2.5 Handling Loops under Symbolic Execution

Occurrences of loops in a program complicate the verification process. We describe how the technique of symbolic execution handles loops. Loops are essentially multiple 'if ... then ...' statements with no way of knowing exactly how many statements are needed because of the loop condition (lc). There are however, two possible methods for handling them: using induction on the loop or using the 'loop invariant' rule. For further information on the first method, induction on loops, refer to [38]. We now take a closer look at the invariant rule. This is a method implemented by KeYmaera.

The idea of this rule is to add as much information to the proof as possible and then continue with normal symbolic execution. Similar to the occurrence of an 'if ... then ...' statement the loop invariant rule splits our program up into 3 paths. (Loop invariant method extracted from [6]).

The formula which needs proving has the form:

$$\Gamma \Rightarrow \mathrm{U}\ [\text{while (lc)p}]\phi, \Delta,$$

where Γ are the pre-conditions, U is the set of updates, (lc) is the loop condition, p is the loop body, ϕ are the post-conditions and Δ are some additional information we can use but that are not important now.

Using an invariant splits the proof into three branches:

1. Invariant Initially Valid:

$$\Gamma \Rightarrow \mathrm{U}\ \mathrm{Inv}, \Delta$$

We have to show that the preconditions implies the invariant.

2. Body Preserves Invariant:

$$\text{Inv} \wedge \text{lc} \Rightarrow [\text{p}]\ \text{Inv}$$

We have to show that the invariant holds during execution of the loop.

3. Use Case:

$$\text{Inv} \wedge !\text{lc} \Rightarrow \phi$$

We have to show, that after termination of the loop, the post-conditions hold. So the invariant and the negated loop condition implies the post-condition.

To illustrate this rule, we present a short example.

Example 3 Extracted from [6]
We want to prove a simple program that decrements a variable x to 0, seen previously in Example 1:

```
1  x ≥ 0 ⇒
2      [ while (x > 0) {
3          x = x - 1;
4      } ]
5  (x = 0)
```

So we use $x \geq 0$ as invariant and we get these 3 paths:

1. Invariant initially valid ($\Gamma \Rightarrow \text{U Inv}, \Delta$):
   $x \geq 0 \Rightarrow x \geq 0$

2. Body preserves invariant ($\text{Inv} \wedge \text{lc} \Rightarrow [\text{p}]\ \text{Inv}$):
   $x \geq 0 \wedge x > 0 \Rightarrow x - 1 \geq 0$

3. Use Case ($\text{Inv} \wedge !\text{lc} \Rightarrow \phi$):
   $x \geq 0 \wedge x \leq 0 \Rightarrow x = 0$
   $\rightarrow$
   $x = 0 \Rightarrow x = 0$

We see that all three paths are trivial to solve.

The key for handling loops is an invariant that offers good usable additional information. If we are able to find a fitting invariant, we are able to handle every loop occurring in our proof. Generating more complicated invariants is a difficult process, and is described in [6].

Having looked at some techniques used by KeYmaera, we feel ready to pass our drillstring system into the program's syntax. Program notation d$\mathcal{L}$ and the style of proof will be of upmost importance to begin this process.

Chapter 4

Using KeYmaera

This section starts by looking at the additional software that is needed to allow us to run KeYmaera successfully. We look at how two, previously studied, dynamical systems have been passed into KeYmaera. Looking at the general form of the programs, the syntax used and the correctness properties which can be verified. The first system we look at is the discontinuous example of the bouncing ball which was studied in Section 1.5.3. The second system we re-visit is that of the water-tank example seen in Section 1.5.2. Understanding how these two dynamical systems are passed into KeYmaera as hybrid programs, along with their respective safety conditions, will help us to correctly pass the drillstring system into KeYmaera.

4.1 Additional Software

Before we begin to study specific programs, it is worth mentioning the additional software that is needed to allow full functionality of KeYmaera.

Java runtime environment 1.5 is needed to run KeYmaera. This is because KeYmaera is a Java based program. The runtime environment is available for free download at, (accessed July 2010):

`http://www.oracle.com/technetwork/java/index.html`.

Reduce and Redlog are additional (and optional) programs which can add proving power once lanced to KeYmaera. For the best proving power however, the algebraic solver Mathematica is recommended. We installed and lanced Wolfram Mathematica 7 for Students to KeYmaera. This gives us the best possible chance of verifying safety critical parameter regions of our system. If Mathematica is not available, the differential solver Orbital can often produce the same results.

For any of the programs needed or recommended, links can be found on the KeYmaera download webpage, (accessed July 2010):

`http://symbolaris.com/info/KeYmaera-download.html`.

4.2 KeYmaera Examples

As previously mentioned, we intend to study two KeYmaera example programs before moving onto create our own programs. We believe that the drillstring system can be passed into a KeYmaera program in a similar way to the dynamical systems in these two example programs.

A selection of example KeYmaera programs, written by the software developers, are included with the package download. The complexity of the example programs range from simple mathematical proofs to the safety verification of complicated hybrid systems. The idea of these programs is to help KeYmaera users understand the verification tool and to get an idea of the formal properties and problems that can be verified with it.

4.2.1 Structure of `.KEY` Files

The two example programs we look at in this section both have very similar structures. We feel it is important to investigate the similar structural aspects before we investigate the details of either of them further.

The two example programs in question are `.KEY` input files for verifying safety properties in KeYmaera. A `.KEY` file specifies both the operational model of the hybrid system, using program notation for hybrid systems previously seen in Section 3.2.2, and the formal property that wishes to be verified. This is the approach we take when creating a `.KEY` input file for the drillstring system.

All KeYmaera programs are made up of what is called a 'verification problem'. It is specified in a block called `\problem{...}` and contains the full description of the problem or system being passed into KeYmaera. For example, in the block there are three main parts:

- The initial state of the hybrid system.
- The operational model of the system described using hybrid program notation, d$\mathcal{L}$.
- The property wished to be verified using modalities. This can be either a safety, liveness, reactivity or controllability property.

The verification problem in both example programs has the form:

$$\backslash[\text{Variable declaration}\backslash]$$
$$(\text{Initial conditions}) \quad -> \quad \backslash[\alpha\backslash](\phi).$$

This reads: Using the variables declared and a particular set of initial conditions, all states reachable by hybrid program α, satisfy the post-condition or safety region ϕ.

This is the basic structure of many example KeYmaera programs which aim to verify a safety condition of a hybrid system. We understand the drillstring `.KEY` input file needs to be of a similar structure. Next we look more closely at the discontinuous bouncing ball example KeYmaera program.

4.2.2 Bouncing Ball Program

Recall that the dynamical system representation of a bouncing ball is a discontinuous system, which evolves with continuous dynamics, with discrete jumps in the velocity vector due to the impact of the ball with the floor at $h = 0$. We studied this system in detail and simulated it in Section 1.5.3. Now however, we intend to thoroughly examine the example KeYmaera program to understand how it verifies the safety condition.

The example code is attached in Appendix D.1. We now take a detailed look at each section of the verification problem.

Variable Declaration

First, the variables that are needed within the program are declared. This is broken up into two sections for state and auxiliary variables. State variables are those needed to keep track of the current state of the system, they are enough to describe the future behaviour of the system. The state variables in this example are h = current height of ball, v = current velocity of ball and t = time taken since the ball was released. Also declared here are any auxiliary variables needed in the proof, c = elastic dampening factor at floor ($h = 0$), g = acceleration due to gravity, H = height limit and V = velocity limit.

Initial Conditions

The declarations are followed by specifying the initial conditions of the system. In this program the hybrid system is initialized with initial height, $h = 0$ and initial velocity, $v = 16$.

Hybrid Program

The system being studied is then defined as a hybrid program inside a box-operator modality. This is where the hybrid system's dynamics, transition

guards and resets are defined in the program. In this example, motion dynamics are specified for h and v. A discrete transition of v represents the discontinuous jump in dynamics when the ball impacts the floor, this also triggers a reset condition on the time variable t:

Listing 4.1: Extract from bouncing ball example program. Appendix D.1

```
1 ->
2   \[ t:=0 ( { h' = v, v'=-10, t'=1, h>=0 }
3         if (t>0 & h=0) then
4             v:=-v/2; t:=0
5         fi )*
6   \]
```

In line 2 of the extract, the motion dynamics are set: $\frac{\mathrm{d}h}{\mathrm{d}t} = v$ and $\frac{\mathrm{d}v}{\mathrm{d}t} = -10$. These specify the motion on the system's state vectors, where the acceleration due to gravity has been rounded to $-10\mathrm{m/s}^2$. The time variable is given a constant rate of change using the notation: $t' = 1$.

To accompany the continuous dynamics in the only location of the representation, there is an invariant condition which must remain true throughout the system's evolution, that is, $h \geq 0$. Invariant conditions on continuous dynamics are optional, as we saw in Table 3.1. However, invariant regions must remain true whilst the system is evolving under the continuous dynamics, otherwise a discrete transition must occur.

When this invariant becomes false, i.e. $h = 0$, the 'if ... then' statement on line 3 is tested. This statement checks whether the discontinuous jump in the velocity vector must be taken. When this discrete transition occurs the time variable is also reset to zero, this enables t to capture the time between each bounce.

The invariant in this modality is optional:

```
1   )*@invariant(h=5*t^2+v*t & h>=0 & t>=0 & v<=-10*t+16 & t<=16/5)
```

These are simple conditions which must hold at all times during the execution of the system. They help with the verification of other properties within the program. Note that this particular invariant is made up of equations of motion and conditions specifying that the ball never drops below the level of the floor and that time must go forwards.

Safety Condition

Here the safety condition ϕ is specified. In this program the condition is:

$$(0 \leq h \leq 13).$$

That is, the height of the ball never drops below zero and it never goes above the height limit, 13. When the program is passed into KeYmaera this is the property it wishes to prove about the system.

It is worth noting that if the hybrid program was specified within diamond box operators, the condition ϕ becomes a liveness property. This is explained previously in Section 3.2.2.

Results in KeYmaera

The example KeYmaera program for the bouncing ball proves the safety condition within seconds. Figure 4.1 is a screen shot of the KeYmaera interface once the proof has been completed.

The original proof can be seen highlighted in green in the middle of the application window. In the left hand proof bar, the three steps to prove loop induction (given in Section 3.2.5) can be seen, each highlighted green showing a successful step. It can therefore be taken that with the initial conditions of the program, the safety condition, $0 \leq h \leq 13$, holds after execution of the

hybrid program.

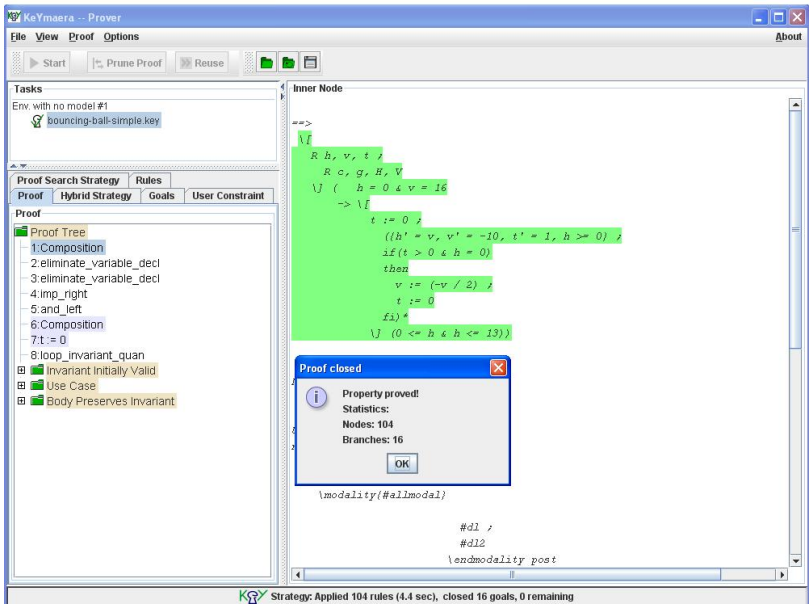

Figure 4.1: Screen shot of KeYmaera interface with the bouncing ball example program, post-proof.

If KeYmaera is successful, proving the hybrid system remains in the safety region after all executions, the 'proof closed' box appears which shows the details of the proof.

4.2.3 Water-Tank Program

The second example KeYmaera program we study, before moving onto develop the drillstring program, is the automated water-tank program. It is based on the four-location automata of the water-tank hybrid system previously seen in Section 1.5.2.

The example program can be seen in Appendix D.2. The verification problem has a very similar structure to the one in the bouncing ball program, so

we will not look at each of the sections in detail again.

However, in the hybrid program a state variable is used to specify the current discrete location of the hybrid system. This technique is very appealing and something we will utilize in the drillstring program. The extract below shows how the first two locations of the system are entered into the hybrid program. We investigate how they are defined in d$\mathcal{L}$ and how the trajectory evolves within the program:

Listing 4.2: Extract from water-tank example program. Appendix D.2

```
1   ->
2     \[
3       (   (?(st=0);
4                 (?(y=10); x:=0; st:=1)
5               ++ {x'=1,y'=1, y<=10}
6           )
7        ++ (?(st=1);
8                 (?(x=2); st:=2)
9               ++ {x'=1,y'=1, x<=2}
10          )   )
```

Before we can explain how this program notation works, we first must specify the state variables: y is the current water level of the tank, x is a time variable and st represents the current state or location of the system, with $Q = \{0, 1, 2, 3\} = \{\text{fill, stop, drain, start}\}$. Refer back to Section 1.5.2 for more details.

The notation ?F checks if the formula F holds at current state, and aborts if not. In line 3 of this extract, the program checks to see whether the state variable, st, is equal to 0. If the condition holds, the system is specified by the dynamics of state 0 (fill location). Otherwise this condition is false, the program then aborts and tests the next relevant state assertion in line 7.

If either of the location conditions is satisfied, line 3 or 7, the program continues onto the subsequent line. Included here are more state assertions which test the transition guards to the other locations in the system. These represent the discrete transitions of the hybrid system. If they are not taken however, the trajectory evolves under the continuous dynamics specified between the curly braces on the following line (lines 5 and 9). In both locations an optional invariant region has been specified along with the continuous dynamics. This invariant region needs to be true at all times during the evolution under the current state dynamics, otherwise a discrete transition needs to take place.

For example, the program initially tests whether the current state is zero or not. The program is initialised in this location, so this is true. The program then tests whether $y = 10$. Initially this is false, since the water-tank is initialised with water level zero, $y = 0$. So this state assertion fails and the program aborts to the next line. The system therefore evolves with the dynamics specified in line 5, until the invariant region ($y \leq 10$) is no longer true. At which point, the time variable is reset, $x := 0$, and the system undergoes a discrete transition. The current state variable is changed to $st := 1$. The program then tests to see which location state assertion holds and the program continues in state 1 (stop location).

Written by the KeYmaera developers, we believe this to be a good way of specifying multiple discrete locations, each with their own continuous dynamics and discrete transitions. We apply this structure to our drillstring program.

4.3 Drillstring System in KeYmaera

Finally we are at a stage where we can be confident passing the drillstring system, introduced in Chapter 2, into a KeYmaera input file. We explain in

detail how each part of the .KEY file is built. Initially we focus our efforts on passing the three location automaton representation of the system, seen in Section 2.5.1, into a KeYmaera verification problem. Figure 4.2 re-visits the automaton here:

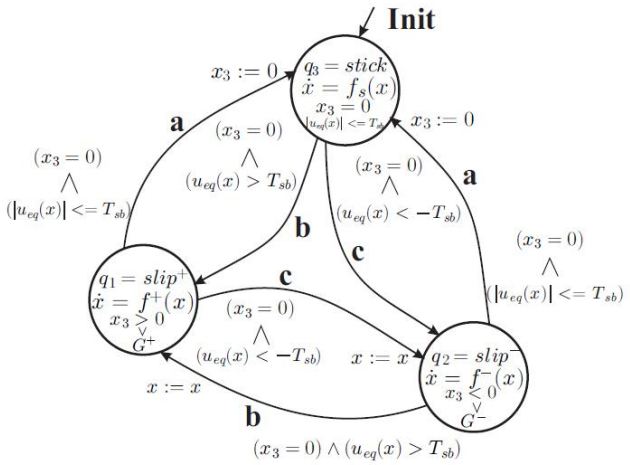

Figure 4.2: Directed graph associated with the three location automaton of the drillstring example [23].

As mentioned, we base the structure of our drillstring program on the .KEY structure outlined above in Section 4.2.1. We recall that a verification problem of the form:

$$\begin{aligned} &\backslash[\text{Variable declaration}\backslash] \\ &\quad (\text{Initial conditions}) \quad -> \\ &\qquad\qquad\qquad\qquad \backslash[\alpha\backslash](\phi), \end{aligned}$$

implies that after all executions of the hybrid program α the system remains in the region ϕ.

We therefore need to declare all necessary variables and initial conditions for the drillstring system, then pass the hybrid system into a modal operator using program notation. Finally, a safe location domain is entered after the

hybrid program and will act as the post-condition we need to verify. This will allow us to prove safe regions of parameters for the drillstring system.

Variable Declaration

It is crucial we declare all variables used to specify the hybrid system in this section. The state variables needed to capture the behaviour of the system are the vector $\boldsymbol{x}$, which is input separately as $x1$, $x2$ and $x3$, along with the variable st for 'current state', or 'current location', of the system within the automata representation.

We specify the value of the two control inputs, $T_m = u$ and W_{ob}, as variables in the auxiliary variable declaration. These appear along with all the other variables needed in the program:

```
1 R Jr, Jb, cr, cb, ct, kt, u, musb, mucb, Rb, Wob, delta, gamb,
2 Vf, Tsb, Tcb, Ueq, Teb;
```

In the subsequent lines of code after the above extract, each auxiliary variable is given an exact value that cannot change throughout the execution of the program. The values of these variables are the same as those we used to simulate the system in Section 2.3.

It is worth mentioning that at the end of the variable declaration section, we define two functions, `Teb` and `Ueq`. To do this we make use of the auxiliary function block, `\functions{...}`. This appears before the `\problem`-block and defines the functions we need in the program. Each is declared using its return type, name and argument types followed by a semi-colon. i.e. `R Ueq(R,R,R);` defines a function, Ueq, with three real input arguments, and an output value of real type.

The use of functions makes the later program notation easier to understand and follow. Also defined for ease are the static and Coulomb friction con-

stants, `Tsb` and `Tcb`. We do this by simply creating a new auxiliary variable equal to the appropriate variables multiplied together:

Listing 4.3: Extract from variable declaration section. Appendix E.1.

```
1  Teb:=(ct*(x1-x3)+kt*x2-cb*x3);
2  Ueq:=(ct*x1+kt*x2-ct*x3-cb*x3);
3  Tsb:=Wob*Rb*musb; Tcb:=Wob*Rb*mucb;
```

Hybrid Program: \[α\]

We use the three discrete locations of the automata as states in the program, shown in Figure 4.2. Lets recall the three location domains, giving each a value of the st variable:

$$\begin{aligned} st := 1 \quad &\Rightarrow Dom(slip^{+}) = \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 > 0\} \cup G^{+}, \\ st := 2 \quad &\Rightarrow Dom(slip^{-}) = \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 < 0\} \cup G^{-}, \\ st := 3 \quad &\Rightarrow Dom(stuck) = S_s^b = \{\boldsymbol{x} \in S_0^b : |U_{eq}(\boldsymbol{x})| \leq T_{s_b}\}, \end{aligned} \tag{4.1}$$

where,

$$\begin{aligned} G^{+} &= \{\boldsymbol{x} \in S_0^b : U_{eq} > T_{s_b}\}, \\ G^{-} &= \{\boldsymbol{x} \in S_0^b : U_{eq} < -T_{s_b}\}, \\ S_0^b &= \{\boldsymbol{x} \in \mathbb{R}^3 : x3 = 0\}. \end{aligned} \tag{4.2}$$

S_0^b is known as the switching surface and S_s^b is the sliding set.

From the location domains it is clear we need three distinct routes through our program, one for each location of the automata. Each route or section of the program must then only be entered when the system satisfies a current location condition, created using state assertions on the variable st.

In each case, if the formula following the state assertion is correct the program continues on to the dynamics specified by that location. If however, the state assertion is wrong, the program continues to the next. But only ever taking one of the three routes. For example:

```
1     ?(st=3); ...
2   ++ ?(st=2); ...
3   ++ ?(st=1); ...
```

From the automata we see that both the other locations are reachable from each of the states via a discrete transition. Therefore under each of the state assertions on the variable *st*, two further state assertions are needed to represent these discrete transitions.

The formula of these state assertions will be the guard conditions of the transitions to the other locations. Therefore the system will only evolve under the continuous dynamics of the current state if neither of these conditions are satisfied and hence no transitions occur. The final piece of code needed within each state assertion on *st* is a set of curly braces defining the current state continuous dynamics, with an optional invariant region.

Recalling the transitions and guard conditions of the drillstring system detailed in Section 2.5.1. For each location of the automata we assemble the two discrete transitions along with the guard conditions, resets and final locations. i.e. For the stuck location, the guard condition to move to slip$^+$ location is:

```
(x3 = 0 & Ueq(x1,x2,x3) > Tsb).
```

This guard condition therefore becomes the first of the two state assertion formulae under the stuck location state assertion: $?(st = 3)$.

We build all six of the transitions up in this way and the following foundation of our drillstring program is achieved:

Listing 4.4: State assertions on the location variable *st* and on the discrete transitions within each location.

```
1     ?(st=3);
2           ?(x3 = 0 & Ueq(x1,x2,x3) > Tsb); x:=x; st:= 1;
3         ++ ?(x3 = 0 & Ueq(x1,x2,x3) < -Tsb); x:=x; st:= 2;
4             ...
5   ++ ?(st=2);
6           ?(x3 = 0 & Ueq(x1,x2,x3) > Tsb); x:=x; st:= 1;
7         ++ ?(x3 = 0 & |Ueq(x1,x2,x3)| <= Tsb); x3:=0; st:= 3;
8             ...
9   ++ ?(st=1);
10          ?(x3 = 0 & Ueq(x1,x2,x3) < -Tsb); x:=x; st:= 2;
11        ++ ?(x3 = 0 & |Ueq(x1,x2,x3)| <= Tsb); x3:=0; st:= 3;
12            ...
```

When a discrete transition takes place, either `x:=x` or `x3:=0` represent the reset condition which is enforced. It is worth noting, that in the actual program, the reset condition of vector $\boldsymbol{x}$ needs splitting up into `x1:=x1; x2:=x2; x3:=x3;`.

We also did not use the absolute value notation in KeYmaera. To represent `|Ueq(x1,x2,x3)| <= Tsb`, the actual program uses the two inequalities:

```
Ueq(x1,x2,x3) >= -Tsb & Ueq(x1,x2,x3) <= Tsb.
```

Finally, the third line under each location state assertion specifies the continuous dynamics of the current location. That is, the dynamics to follow if the system does not take either of the discrete transitions. We include a further state assertion before the dynamics, which will hold if neither of the discrete transitions are taken.

Recall the dynamics from Section 2.5.1. For location slip$^+$, slip$^-$ and stuck we use $f^+(\boldsymbol{x})$, $f^-(\boldsymbol{x})$, and $f_s(\boldsymbol{x})$ respectively. We add these dynamics in

curly braces below the state assertions which represent the discrete transitions. This concludes the building of our modal operator and program, \[α\].

In summary we have three separate locations or routes through the program, each with three lines of program code. Two describing the possible discrete transitions from the current location and the final line representing the current location continuous dynamics. Below is an extract showing the three lines of code representing the stuck location section of the program:

```
1 (?(st=3);
2     (?(x3 = 0 & Ueq > Tsb); x1:=x1; x2:=x2; x3:=x3; st:=1)
3   ++(?(x3 = 0 & Ueq < -Tsb); x1:=x1; x2:=x2; x3:=x3; st:=2)
4   ++(?(x3 = 0 & Ueq >= -Tsb & Ueq <= Tsb); {x'=f_s(x), x3=0} ) )
```

Note that in the actual program, Ueq is a function and therefore is called using the notation `Ueq(x1,x2,x3)`. Also the dynamics specified in the sliding set, `x' = f_s(x)` must be entered separately for each of the state vectors `x1`, `x2` and `x3`.

Safety Condition: ϕ

This is where we insert the safety condition we want verifying. We understand that a safe drillstring system has positive velocity at the bit rotary inertia at all times. So we begin with that, a safety post-condition specifying that we remain in the positive velocity location:

```
(x3 > 0 | (x3=0 & Ueq(x1,x2,x3) > Tsb)) .
```

This is the domain of the slip$^+$ location and given the appropriate control parameters it is where our system hopefully should remain.

We have explained how each section of the verification problem for our drillstring system has been created. Showing in detail how the system is passed

into KeYmaera program notation. The complete program is attached in Appendix E.1. In the next chapter we set to analyze, test and improve this program.

Chapter 5

Analysis of Drillstring Program

The drillstring KeYmaera program created in the previous chapter is based upon the three state automaton representation of the drillstring system. We passed the hybrid system into hybrid program notation using a verification problem which includes variable declarations, modal operators and a safety post-condition. We are aware that this may not be the most efficient method to verify a safe system using certain parameters in KeYmaera. We feel however that the `.KEY` input file created is simple enough to understand and manipulate. We embark on some initial testing and further development of the program in this chapter.

5.1 Initial Testing

We begin by setting some initial tests for our drillstring program. These are intended to check the program is functioning as expected. By using the same safety post-condition, but altering the control parameters, we can see if our KeYmaera program is successful in proving that after all executions of the hybrid program, the system remains within the safety region.

From simulations of the system in Section 2.3, we already know values of the control parameters u and W_{ob} which produce each of the three types

of long term behaviour. We therefore fix $u = 4300Nm$, and by altering $W_{ob} = 35500N$, $36000N$ and $41000N$ we know the system exhibits convergence to equilibrium, stick-slip and stuck long term behaviours respectively. These parameter values become our three test cases, for which we know the system's long term behaviour, and therefore can expect certain KeYmaera outcomes.

Test 1: Using the domain of the positive slip location as a safety post-condition. Our KeYmaera program should prove one out of the three cases presented. The stick-slip and stuck bit cases however, should fail because the system does not stay within the necessary safety region for all executions of the program.

Test 2: We also test the more simple safety post-condition of a non-negative rotary bit, $x_3 \geq 0$. This must hold at all times during the execution of the program. We use the same control parameter cases as the previous test and from the simulations in Section 2.3. Therefore, we know that the convergence to equilibrium and the stick-slip behaviour satisfy this post-condition for all executions, but the stuck behaviour case should fail due to the dynamics seen in Section 2.3.2.

These are the first two initial tests of the drillstring program. In summary, it includes two different safety post-conditions with three control parameter cases each. We know the simulated behaviour of all the cases presented and therefore are simply trying to verify our program works correctly.

For these preliminary tests, we use initial conditions specified by the stuck location. This is because the automata representation we have based our program on, Figure 2.7, is initialised in this location. It is also the same initial conditions which were used when simulating the system in Section 2.3. Therefore, we know the long term behaviour of the exact system which uses

these parameter values and we use this knowledge to predict the outcome of the KeYmaera proof.

We set the initial conditions to $\boldsymbol{x} := (0,0,0)$ and $st := 3$ and insert the two safety conditions:

```
(x3 > 0 | (x3 = 0 & Ueq > Tsb))   Positive slip invariant,
        (x3 >= 0)                  Non-negative rotary bit invariant,
```

one at a time after the modal operator square braces. Note: the notation | in the positive slip location invariant means '*or*'.

Figure 5.1 contains a screen shot of the drillstring program loaded in KeYmaera. It is the case, awaiting proving, of $u = 4300Nm$ and $W_{ob} = 35500N$, with the positive slip location as post-condition:

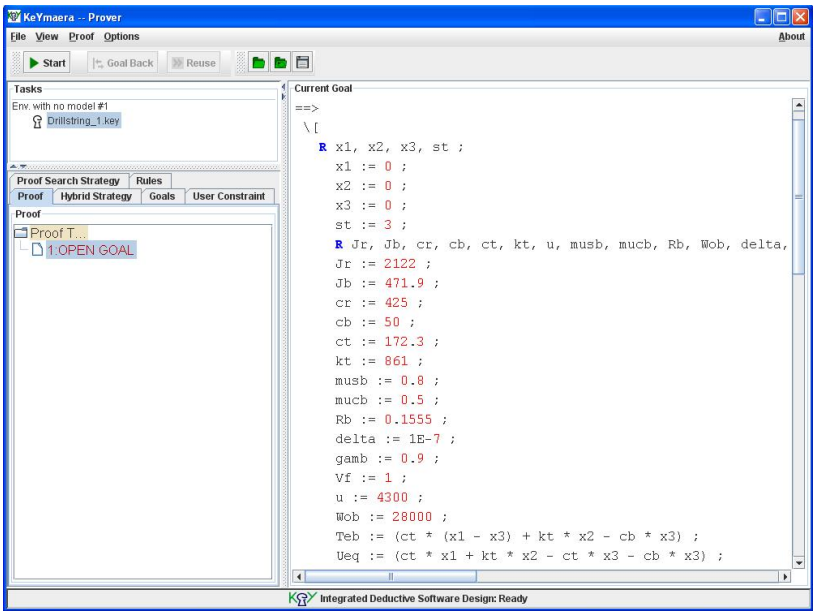

Figure 5.1: Screen shot of KeYmaera interface with the drillstring program loaded, pre-proof. Visibly showing the state and auxiliary variable declaration.

In Figure 5.1 we can clearly see the state variables declared first, followed by the complete list of auxiliary variables, each given an exact value. The modal operator or hybrid program follows the declarations which continues off the bottom of the application window. The post-condition of the positive slip location is defined after the modal operator at the end of the program.

5.2 Initial Results

We tested our drillstring program with the two different safety post-conditions on the three control parameter cases. After comparing the KeYmaera proof outcomes, with the knowledge gained simulating the system in Section 2.3, we obtain the following table of results:

Testing of program: positive slip location as safety				
u(Nm)	W_{ob}(N)	Expected outcome	KeYmaera outcome	Notes
4300	35500	Prove	Proof Failed	Unexpected
4300	36000	Fail	Proof Failed	As expected
4300	41000	Fail	Proof Failed	As expected

Table 5.1: Results obtained when testing the three cases of control parameters which produce each of the behaviours, using the positive slip location safety region.

Secondly, using the simple non-negative rotary bit post-condition as a safety condition after the hybrid program we obtain the set of results in Table 5.2.

From the results in Tables 5.1 and 5.2, we see there is obviously a problem with our drillstring program. The worrying trend is that all the proofs failed, even the parameter cases where we know the system exhibits the long term behaviour to remain within the specified safety post-condition. We proceed by testing the program thoroughly. We attempted to verify the system remains in either of the safety post-conditions using many different control

Testing of program: non-negative rotary bit safety condition				
u(Nm)	W_{ob}(N)	Expected outcome	KeYmaera outcome	Notes
4300	35500	Prove	Proof Failed	Unexpected
4300	36000	Prove	Proof Failed	Unexpected
4300	41000	Fail	Proof Failed	As expected

Table 5.2: Results obtained when testing the three cases of control parameters which produce each of the behaviours, using a non-negative rotary bit safety.

parameter pairings. We obtained the pairings from the simulations carried out by R. Carter in [10], so we know the predicted KeYmaera outcome. Again all the proofs failed, even for parameter values which we know the system converges to an equilibrium.

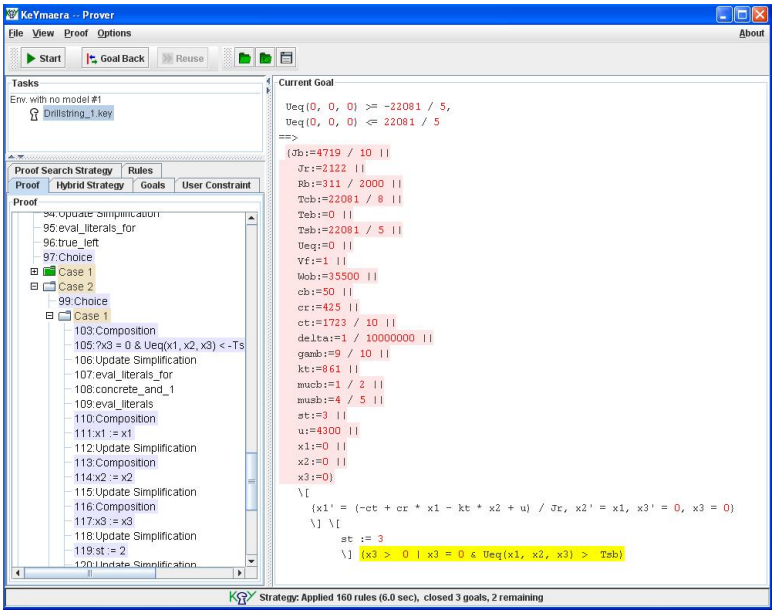

Figure 5.2: Screen shot of KeYmaera application window, with parameters $u = 4300Nm$ and $W_{ob} = 35,500N$ using a positive slip location invariant (highlighted), post-proof.

Figure 5.2 is a screen shot of the KeYmaera application window after a proof has failed. It is the particular case of $u = 4300Nm$ and $W_{ob} = 35500N$, trying to verify the system remains in the positive slip location. This is one of the unexpected failures because the simulations carried out with these parameters in Section 2.3, showed this system to converge to an equilibrium and hence remain in the positive slip location.

This concludes our initial testing of the drillstring program. In summary we saw KeYmaera fail to verify any of the initial tests we set. Our conclusion is that, even when using control parameters that we know produce positive bit velocity, the drillstring program in KeYmaera does not remain in either of the two safety regions we tested. The only explanation is that the programs, with parameters which were expected to remain within the safety regions, for some reason do not, and the proofs in KeYmaera were not successful.

In the next section we begin to analyse the KeYmaera outcomes to find reasons the testing did not go as expected. We look at potential problems we face and possible modifications to the program.

5.3 Problems Encountered

In this section we try to outline the reasons KeYmaera failed and why it believes the hybrid programs, which are only a representation of the system, strayed from the safety region rendering the proofs false. From simulating the system, either KeYmaera must be malfunctioning or we are implementing the drillstring system in KeYmaera incorrectly. Once we identify possible problems with our program, we implement some modifications in Section 5.4.

5.3.1 Initialization Problem

When we expected a successful proof, but obtained a fail case, the KeYmaera proof structure highlighted one particular problem. It is shown in Figure 5.2 and appeared multiple times during the initial testing. It is the problem:

```
Ueq(0,0,0) >= -Tsb
Ueq(0,0,0) <= Tsb  ⇒
                        {Stuck location continuous dynamics}
                        {Post-condition}.
```

Imagine the program KeYmaera is attempting to prove, uses the positive slip location as the safety post-condition,

```
(x3 > 0 | (x3 = 0 & Ueq > Tsb)),
```

we can clearly see that it does not hold. More importantly however, is that the post-condition fails on initialisation of the program.

If we initialise in the stuck location with $\boldsymbol{x}$:= (0,0,0), the system is outside the safety region (positive slip location) initially. This is obviously a fault with our program. There are two possible solutions to this problem.

1. Make sure the safety post-condition or invariant is true on initialisation. We can achieve this in two ways:

- By initialising the system in another location other than stuck. i.e. initialising the state vector $\boldsymbol{x} > (0, 0, 0)$ and `st:=1`. This would ensure that both the positive slip location and the non-negative bit velocity post-conditions hold initially.
- Alternatively, we could make the condition `(Ueq > Tsb)` hold initially. This would allow us to leave the state vector initialised at $\boldsymbol{x}$ `:= (0,0,0)`.

However, there are problems with applying either of these methods. If we are basing our expected behaviour on the MatLab simulations undertaken in

Section 2.3, we cannot change the initial conditions of the drillstring program in KeYmaera and continue to compare the outcomes to the simulations. The reason behind this is that the simulations were initialised at (0, 0, 0), and by changing the initial conditions of the system, the long-term behaviour could differ.

We are also unable to alter either the function, Ueq, or the static friction constant, Tsb, initially, without it having further implications on the evolution of the system.

2. A second approach considered is to create a timed hybrid program. Timed hybrid programs contain a clock variable which evolves at a constant rate, `t':=1`. This ideally would allow us to test invariants or safety conditions, after a certain amount of time had elapsed. For example, to test whether the bit rotary system has non-negative velocity for all time after 100 seconds, we could use the safety post-condition:

```
t > 100 -> x3 >= 0.
```

The program could still be initialised in the stuck location and we could still obtain expected KeYmaera outcomes from the comparison with the MatLab simulations.

This method was initially explored to tackle our problem, but we will not progress with a timed program. There are several reasons for this, one being that the hybrid automata representations we are basing our hybrid programs on, do not use a time variable. There is also a distinct lack of example programs which use a clock variable as part of their safety post-condition. Many example programs use them as simple invariant conditions, inside the modal operators, i.e. $t > 0$, which means time cannot evolve backwards. But none of the developer written example programs attempt to use a clock variable to verify safety conditions, on a time restricted portion of a system's behaviour. It may well not be possible.

The other main argument against using a clock variable to test the safety condition after some transient, is that it removes the behaviour of the transient from the safety verification process. We neither know how long the transient will be in any given case, or how harmful the system behaviour could be during the transient. This reduces the effectiveness and main reason of verifying a safe system in the first place.

5.3.2 Function Problems

During the initial testing and in our attempts to overcome the initialisation problem, we encountered further issues. These often involved the functions that we were attempting to use within the hybrid program. The main offender was the exponential function used in the bit-to-rock friction model. We encountered two problems regarding this function.

1. An error occurred when we declared the exponential function in the `\functions`-block. This was because `exp` is supposedly a pre- defined function within KeYmaera. However, we have no way of checking this assumption is correct.

2. An error of the form: `exp:R(Sort:R);` occurred while attempting to verify the system using alternate initial, and safety conditions. The sub terms of the error identified the problem. It was caused by the exponential function in the friction term, while the system evolved with either slip^+ or slip^- continuous dynamics.

We have already prepared for such problems with the exponential function, and have detailed alternative friction models we can implement instead. We propose variants of the original drillstring program in the next section, making use of the alternative methods.

As well as problems with the exponential function, it is worth noting that there is no guarantee any of the functions we define in the \functions-block evolve correctly. We restrict the number of functions in our drillstring program, we only use the Exponential function, Ueq(x_1, x_2, x_3) and Teb(x_1, x_2, x_3). These are integral to the evolution of the system. The functions are used in the state assertion conditions as guard conditions restricting/allowing discrete transitions to take place along with in the continuous dynamics of each location.

It is worrying that there is no way to check if the functions in the program are being used correctly in the KeYmaera proof. We cannot seek out intermediate values as we can do with simulation software, as addressed in the next section.

5.4 Variants of the Drillstring Program

We investigated all the problems highlighted in the previous section. Possible alternative implementations, which each create a variant to the original drillstring program, were considered. We present each new implementation with new initial testing. Three parameter cases, were simulated beforehand with MatLab, to obtain expected behaviour. The aim is to find a program implementation which fairs better in the comparison to the MatLab simulations than the original.

5.4.1 Initial Conditions

First we must clarify that we can only use the MatLab simulations as expected results for the KeYmaera output, if we are attempting to prove properties about exactly the same system as we simulated.

As mentioned in Section 5.3.1, the initial conditions can cause the safety condition to be false when initialising the drillstring program in KeYmaera.

We understand that the initial condition $\boldsymbol{x} := (0,\ 0,\ 0)$ can not work when using a positive slip location as a safety post-condition.

Therefore, if we wish to use this safety condition, new initial conditions for the system must be used. The system must be simulated again using MatLab, and the new results become the expected KeYmaera outcome to confirm/dismiss that our program is working properly. The new initial conditions agreed upon:

$$\boldsymbol{x} := (0.1, 10, 0.1) \texttt{ \& st:=1},$$

are to be used when the safety condition would fail instantly due to the initialisation. This gives the system a chance of remaining in the post-condition within the KeYmaera proof, and not just of failing instantly.

For a fixed $u = 3500Nm$ and a range of parameter $W_{ob} \in [2900, 35000]$, we compared the simulations presented by R. Carter in [10], to those simulated using the new initial conditions. We obtained exactly the same behaviour with the new initial conditions over a timespan of [0, 100]. However, KeYmaera attempts to prove a safety post-condition for all executions of a system, it is not time restricted. Hence, we ran the simulations for a longer timespan, and obtained two cases where the behaviour appears stick-slip within [0, 100], but eventually converges to an equilibrium. These were the cases of $u = 3500Nm$ with $W_{ob} = 30100N$ and $30250N$.

We now have expected behaviour for the drillstring program initialised with the new initial conditions. Table 5.3 shows a comparison of these simulated behaviours against the KeYmaera outcomes for the drillstring program initialised with the same values. We test on the same range of parameters and try to verify the positive slip location post-condition:

Testing of original drillstring program with $\boldsymbol{x}$:= (0.1, 10, 0.1)				
u(Nm)	W_{ob}(N)	Simulated Behaviour	KeYmaera outcome	Notes
3500	29000	Pos vel.	Proved	As expected
3500	30000	Pos vel.	Proved	As expected
3500	30050	Pos vel.	Proved	As expected
3500	30100	Pos vel.	Proved	As expected
3500	30250	Pos vel.	Proved	As expected
3500	30500	Stick-slip	Proved	Unexpected
3500	31000	Stick-slip	Proved	Unexpected
3500	32500	Perm. Stuck	Proved	Unexpected
3500	35000	Perm. Stuck	Proved	Unexpected

Table 5.3: Results obtained when testing the original drillstring program initialised in the positive location, using the positive location safety condition.

Again there is a worrying trend. KeYmaera proves that the system remains within the positive slip location (safety condition) for all executions, even when using control parameters that we know produce a permanent stuck bit.

We conclude from these tests that KeYmaera is not understanding the hybrid system evolution properly. KeYmaera works with our hybrid program that is a representation of the hybrid system detailed in Chapter 2. If we initialise the program in the positive slip location, and the program is not evolving correctly, it would never escape from this location. Hence it would prove this safety post-condition true for all control parameter cases.

However, we see this as a more manageable problem to fix than the initialisation problem. Nothing can be done with the program if we continue to initialise it in the stuck location, where the safety instantly fails.

We continue to use the new initial condition, and proceed by attempting

to find an alternative implementation that allows our KeYmaera program to evolve as expected. The new implementations to the program also aim to remove the minor problems, detailed in Section 5.3.

5.4.2 Alternative Implementations

We investigated many alternate ways to model the bit-to-rock friction torque in Section 2.4. Due to problems with the exponential function present in the current drillstring program, we take the opportunity in this section to see if an alternative friction model is more successful. The idea being, if we remove the original exponential-type friction and replace it with a less problematic method, it might yield the expected KeYmaera outcomes.

Taylor Series Approximation

Many of the problems previously mentioned involve functions, namely the exponential function. Therefore the most obvious alternate friction model to implement first is the Taylor Series approximation, looked at in Section 2.4.1.

This KeYmaera program is heavily based on the original drillstring program, which uses exponential-type decaying friction, and is attached in Appendix E.1. However, there are minor alterations needed for the program to accept this friction model implementation. These are:

- In the \ `function`-block, we must define the function `R Tay(R);`. This tells KeYmaera that Tay is a function with one real argument and real output.
- The new function Tay is included and defined in the list of auxiliary variables. It is defined as: `Tay:= (1 + x + x^2/2 + x^3/6 + x^4/24);`.
- Finally, the continuous dynamics for x_3 in the slip$^+$ and slip$^-$ locations are changed to:

```
1 x3'=(Teb(x1,x2,x3)-(Tcb+(Tsb-Tcb)*Tay((-gamb/Vf)*|x3|)) )/Jb .
```

However, this method of implementing the Taylor Series approximation still makes use of a function call, in a similar way to the original program.

Alternatively, the implementation can be obtained without, the use of potentially problematic, function calls. We know that:

$$exp^{(\gamma_b|x_3|/v_f)} \approx (1 + (-\gamma_b|x_3|/v_f) + \tfrac{1}{2}(-\gamma_b|x_3|/v_f)^2 + \tfrac{1}{6}(-\gamma_b|x_3|/v_f)^3 + ... \\ ... + \tfrac{1}{24}(-\gamma_b|x_3|/v_f)^4)$$

The continuous dynamics for x_3, in the slip$^+$ location of the KeYmaera program, can therefore be approximated by:

```
1 x3' = (Teb(x1,x2,x3)- (Tcb+(Tsb-Tcb)*( 1 + (-gamb*x3/Vf) +
2       (-gamb*x3/Vf)^2/2 + (-gamb*x3/Vf)^3/6 + (-gamb*x3/Vf)^4/24)))/Jb;
```

Similarly for slip$^-$, remembering however that in the slip$^-$ location $x_3 \leq 0$, and to obtain the absolute value, $|x_3|$, `(-x3)` must be entered in the place of `x3`.

In Section 2.4.1 we simulated this system and compared it to the original drillstring system. Table 2.1 shows these comparisons, however they were created using initial conditions $\boldsymbol{x} := (0, 0, 0)$. Therefore, we must once again re-simulate with positive initial conditions. We find that the three cases of $u = 3500Nm$ with $W_{ob} = 29000N$, $31000N$ and $35000N$, maintain the same long-term behaviour. They exhibit convergence to an equilibrium, stick-slip and permanent stuck bit behaviour respectively. These become our three initial test cases for this new implementation.

Using the positive slip location as the safety post-condition once again, the drillstring program with Taylor Series approximation to the exponential function in the friction, was tested. The three control parameter cases outlined were taken, and we obtain the table of results:

Testing the Taylor Series approximation program				
u(Nm)	W_{ob}(N)	Simulated Behaviour	KeYmaera outcome	Notes
3500	29000	Pos vel.	Proved	As expected
3500	31000	Stick-slip	Proved	Unexpected
3500	35000	Perm. Stuck	Proved	Unexpected

Table 5.4: Results obtained, testing on the three cases of control parameters which produce each of the three behaviours, using the positive slip location as safety region.

Our problem, that KeYmaera proves the program remains in the positive slip location, clearly still remains. We tested further control parameter pairings and KeYmaera continued to prove the safety post-condition. From simulations, we know that the system does not remain in this region, and hence cannot agree with these results.

In Section 2.4.2 we describe another possible friction torque model. It makes use of a Karnopp friction model with a quotient-type decaying friction torque at the bit. It was implemented in KeYmaera in a similar way to the previous alternate friction implementations. The program again continued to prove the safety condition, given any control parameters.

In summary, the drillstring program initialised in the stuck location, fails instantly when a positive slip location safety condition is attempted to be verified. Otherwise, it is initialised with a positive velocity and KeYmaera believes it remains in the positive slip location for all executions.

Five Location Implementation

A five location automata representation of the drillstring system was described in Section 2.5.2. Because of the problems we face with the three location automata, an implementation of the KeYmaera drillstring program

based on this automata representation is fashioned.

To model the friction, it makes use of a combination of dry, switch and Karnopp's friction model explained in Section 2.4.3. As mentioned, this friction model was proposed in [16], and makes use of the prototype parameters given in [18].

We begin to pass this five location automata into KeYmaera by inserting state assertions on the state variable `st`, which now takes a value 1-5. The necessary guard conditions are then inserted under the relevant state assertions, these describe the discrete transitions. The final line of code in each location details the reset condition and the domain, along with the continuous dynamics of that location. The complete KeYmaera program is attached in Appendix E.2.

We proceed once again with initial testing, and comparing to MatLab simulations to supply our tests with expected outcomes. Using the MatLab code attached in Appendix C.5 and C.6 but with initial conditions of $\boldsymbol{x} := (0.1, 10, 0.1)$, we were able to force the systems to exhibit each of the three long-term behaviours. The parameter cases we use are $W_{ob} = 100N$ with $u = 8.3Nm$, $7.4Nm$ and $7.3Nm$ for convergence to equilibrium, stick-slip and permanent stuck bit behaviour respectively [25]. Remembering that this model uses the prototype drillstring parameters and is not an accurate size representation of a real drillstring.

The KeYmaera outcomes were once again compared to the simulated behaviour. Table 5.5 shows that the five location representation is still effected in the same was as the original drillstring program. Further testing also shows that initialising in the stuck location, as expected, fails instantly.

Testing the five location representation program				
u(Nm)	W_{ob}(N)	Simulated Behaviour	KeYmaera outcome	Notes
8.3	100	Pos vel.	Proved	As expected
7.4	100	Stick-slip	Proved	Unexpected
7.3	100	Perm. Stuck	Proved	Unexpected

Table 5.5: Results obtained, on testing the five location drillstring program initialised in the positive location, using the positive location safety condition.

We felt that by changing the automata representation we base our KeYmaera program on, might help highlight the problem with the program's evolution and possibly even avoid it. This however, is not the case. The five location KeYmaera drillstring program also remains in the positive slip location for all time once it is initialised with positive velocity.

Removal of Functions

After much additional testing of each of our drillstring programs, KeYmaera continued to prove that the system remained in the positive slip location after being initialised with a positive velocity. None of the proposed programs allowed the drillstring system to evolve out of this location. So again we know either KeYmaera must be malfunctioning or we are implementing our drillstring system incorrectly.

Although we make no further major implementations, we notice that all our programs rely on at least two function, Ueq and Teb. We understand that if KeYmaera is misinterpreting these functions, the hybrid program would not evolve properly and would remain in the location it is initialised in. We present an implementation of the drillstring system which does not rely on the use of function calls.

We base this final drillstring program on the original three location automata,

created in Section 4.3. This variant of the drillstring program has been scrutinised the most. The initial testing took place in Section 5.1, where it was compared to MatLab simulations. These simulations also provide the expected KeYmaera outcomes for testing this program. The KeYmaera drillstring program with no function calls is attached in Appendix E.3.

The initial testing took place once again and the problems remained. The system remained incapable of leaving the location it is initialised in, regardless of the control parameters.

The motivation behind the program with no function calls was that if it had been successful, we could have stated that KeYmaera had been using the functions incorrectly in the other programs proposed. This however is not the case. Our KeYmaera programs do not allow the drillstring system to be represented in a way which captures the system's dynamics and to be verified correctly.

5.5 Final Results

Throughout this chapter we have tested each of our KeYmaera drillstring programs comprehensively. We have compared our KeYmaera proof outcomes to the corresponding MatLab simulations for each new implemented change. However, equivalent results between the verification software and the simulations were never achieved on multiple parameter cases.

The initialisation of the system in the stuck location meant a positive slip location safety condition failed instantly. Furthermore, initialising the drillstring with a positive velocity disrupted the evolution of the system, and it verified the positive slip location safety condition for all control parameters. We believe, that KeYmaera did not fully appreciate the dynamics of the system, and that the discrete transitions were never taken as a result.

Chapter 6

Conclusion

In this project we have studied a discontinuous dynamical system, with the aim of formally verifying a safe region of control parameters. We initially simulated the system in MatLab, we used multiple representations and different friction models to gain an understanding of the long-term behaviours the system exhibited.

We investigated formal verification techniques, and showed how they can be used on a dynamical system to verify a reachable set of states. To do this, program notation and logic used in verification software was looked at. This allowed us to pass the discontinuous drillstring system into KeYmaera. We established a basic proof structure which enabled us to test, that a safety condition remains true for all executions of a system under certain initial conditions.

We presented the drillstring system as a KeYmaera program alongside MatLab simulations for comparison. Multiple implementations were attempted to make the KeYmaera proof outcomes agree with the long-term simulated behaviour.

During our investigation into KeYmaera, we discovered that when it tries

to verify properties, it uses a discrete abstraction of the system. Due to the complexity of the drillstring system, we never achieved verification a safe domain. We believe that KeYmaera was not using a representation which captured all the characteristics of the hybrid system.

The only implementation that was not given thorough testing was the representation which introduced a clock variable. It may have enabled us to verify a safety condition on a time restricted period of the system's evolution. i.e. Removing the transient from the evolution of the system and then verifying some safety property. This route however was not taken, since emphasis was on the drillstring representations of the three and five location automata.

Bibliography

[1] Tricone manual de barrenas. *USA: Hughes Tool Company*, 1982.

[2] Bouncing ball simulink model. MathWorks Inc., July 2010. `www.mathworks.com/products/simulink/demos.html?file=/products/demos/shipping/simulink/sldemo_bounce.html`.

[3] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of Lecture Notes in Computer Science, pages 209–229. Springer-Verlag, 1993.

[4] P. Antsaklis, J. Stiver, and M. Lemmon. Hybrid systems modelling and auronomous control systems. In *Hybrid Systems*, volume 736 of Lecture Notes in Computer Science, pages 366–392. Springer-Verlag, 1993.

[5] J. Awrejcewicz and C. H. Lamarque. *Bifurcation and chaos in nonsmooth mechanical systems*. World Scientific Series on Nonlinear Science, Series A vol 45, 2003.

[6] M. Bedner. Finding loop invariants by static program analysis. In *Seminar Programmverifikation*, December 14th, 2006.

[7] P. Bogacki and L. F. Shampine. *A 3(2) pair of Runge-Kutta formulas*, volume 2 pages 1-9. Appl. Math. Letters, 1989.

[8] M. Branicky, V. Borkar, and S. Mitter. A unified framework for hybrid control: Model and optimal control theory. In *IEEE Transactions on Automatic Control, 43(1):31-45*, 1998.

[9] M. Buss, M. Glocker, M. Hardt, O. von Stryk, R. Bulirsch, and G. Schmidt. Nonlinear hybrid dynamical systems: modelling, optimal control, and applications. In *Modelling, Analysis and Design of Hybrid Systems, LNCIS, 279:311-335*. Springer-Verlag, Berlin, 2002.

[10] R. Carter. Computational model of a rotary system with discontinuous elements. Master's thesis, School of Computer Science, The University of Manchester University, UK, 2009.

[11] J. R. Dormand and P. J. Prince. *A family of embedded Runge-Kutta formulae*, volume 6, pages 19-26. J. Comp. Appl. Math., 1980.

[12] T. A. Henzinger. The theory of hybrid automata. In *Proc. of the IEEE Symposium of Logic in Computer Science*, volume 1, pages 278–292. Electrical Engineering and Computer Science University of California at Berkeley, 1996.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages and computation. Addison-Wesley Longman Publishing Co., 3rd Edition, 2007.

[14] J. Huke. The dynamical systems approach to nonlinear signal processing. In *IMA Tutorial Course on Nonlinear and Non-Gaussian Signal Processing*. Department of Mathematics, UMIST, Manchester, December 2009.

[15] D. Karnopp. Computer simulation of stick-slip friction in mechanical dynamic systems. In *ASME Journal of Dynamic Systems, Measurement, and Control*, volume 107,(1):100-103, 1985.

[16] R. Leine. *Bifurcations in Discontinuous Mechanical Systems of Filippov-type.* PhD thesis, Technical University of Eindhoven, The Netherlands, 2000.

[17] R. Leine, D. van Campen, A. de Kraker, and L. van den Steen. Stick-slip vibrations induced by alternate friction models. In *Nonlinear Dynamics*, volume 16:41-54, 1998.

[18] N. Mihajlovic, A. van Veggel, N. van der Wouw, and H. Nijmeijer. Analysis of friction-induced limit cycling in an experimental drill-string system. In *ASME Journal of Dynamic Systems, Measurement, and Control*, 2003.

[19] E. M. Navarro-Lopez. *What makes the control of discontinuous dynamical systems so complex?*, pages 91–116. In Mathematical Problems in Engineering. An International Series of Scientific Monographs and Text Books, Cambridge Scientific Publishers, UK, 2009.

[20] E. M. Navarro-Lopez. Discontinuities-induced phenomena in an industrial application: analysis and control solutions. In *ICNPAA 2008 7th International Conference on Mathematical Problems in Engineering, Aerospace and Sciences, Genoa, Italy*, June 2008.

[21] E. M. Navarro-Lopez. Hybrid modelling of a discontinuous dynamical system including switching control. In *CHAOS09, 2nd IFAC Conference on Analysis and Control of Chaotic Systems, London, UK*, June 2009.

[22] E. M. Navarro-Lopez. Hybrid-automaton models for simulating systems with sliding motion: still a challenge. In *3rd IFAC Conderence on Analysis and Design of Hybrid Systems, ADHS'09, Zaragoza, Spain*, September 2009.

[23] E. M. Navarro-Lopez and R. Carter. Hybrid automata: An insight into the discrete abstraction of discontinuous systems. *International Journal*

of Systems Science. Special issue on Variable Structure systems Methods for Control and Observation of Hybrid Systems, 2010.

[24] E. M. Navarro-Lopez and D. Corts. Avoiding harmful oscillations in a drillstring through dynamical analysis. *Journal of Sound and Vibration*, 2007.

[25] E. M. Navarro-Lopez and R.Suarez. Modelling and analysis of stick-slip behaviour in a drillstring under dry friction. In *Programma de Investigacion en Mathematicas Aplicadas y Computacion Instituto Mexicano del Petroleo, Mexico*, 2004.

[26] A. Platzer. Differential dynamic logic for verifying parametric hybrid systems. In *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 4548 of *LNCS*, pages 216–232. Springer, 2007.

[27] A. Platzer. Differential dynamic logic for hybrid systems. In *Automated Reasoning*, volume 41 of *Lecture Notes in Computer Science*, pages 143–189. Springer Berlin Heidelberg, 2008.

[28] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixed points. *Formal Methods in System Design*, 35(1):98–120, 2009.

[29] A. Platzer and E. M. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In *International Symposium on Formal Methods*, volume 5850 of LNCS, pages 547–562. Springer, 2009.

[30] A. Platzer and J.-D. Quesel. Keymaera: A hybrid theorem prover for hybrid systems. In *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer Berlin Heidelberg, 2008.

[31] M. W. Reichelt and L. F. Shampine. Matlab code: ballode.m. The MathWorks, Inc. Revision: 1.17.4.2, 1995.

[32] L. Tavernini. Differential automata and their discrete simulators. In *Nonlinear Analysis, Theory, Methods and Applications*, pages 11(6):665–683, 1987.

[33] C. Tomlin, I. M. Mitchell, A. M. Bayen, and M. K. M. Oishi. Computational techniques for the verification and control of hybrid systems. In *Multidisciplinary Methods for Analysis Optimization and Control of Complex Systems*, volume 6, Part I, 151-175 of *Mathematics in Industry*. Springer Berlin Heidelberg, 2005.

[34] V. I. Utkin. Automation and remote control. *Volume 44 Number 9, Part 1*, 1983.

[35] V. I. Utkin. Sliding modes in control and optimization. *Springer-Verlag, Berlin*, 2002.

[36] A. van der Schaft and J. Schumacher. An introduction to hybrid dynamical systems. *Springer-Verlag, London*, 2000.

[37] P. Varaiya. Smart cars on smart roads: Problem of control. *IEEE*, pages 195–207, 1993.

[38] A. Wallenburg. Induction rules for proving correctness of imperative programs. Technical report, Department of Computer Science and Engineering Chalmers University of Technology and Goteborg University, Sweeden, 2004.

[39] H. Ye, A. Michel, and L. Hou. Stability theory for hybrid dynamical systems. In *IEEE Transactions on Automatic Control*, pages 43(4):461–474, 1998.

Appendices

Appendix A

Appendix A.1

Matlab DriverScript: Bouncing ball example

```
% MATLAB script file to simulate a vertically bouncing ball
% ODE45 is used to simulate the bouncing ball,
% and results are plotted.

tspan = [0,20];
y0 = [10  0];
[t,y] = ode45(@f,tspan,y0);
%plot(t,y)

subplot (1,2,1)
plot (t, y(:,1),'-r', t, y(:,2), '-b')
title ('Bouncing Ball')
legend ('Height of Ball','Velocity of Ball','Location','Best')
axis auto
xlabel ('Time t')
ylabel ('Solution y')
```

Appendix A.2

MatLab Function Handle: Bouncing ball ode45 dynamics function file

```
function dy = f(t,y)

c = 0.5;
g = 9.8;
Dv = 0.000001;

v = vel;

dy = zeros(2,1);
dy(1) = v;
dy(2) = -g;

    function v = vel
        bounce = -c*y(2);
        falling = y(2);

        if y(1) > Dv
            v = falling;
        else
            v = bounce;
            y(1) = 0;
            y(2) = -c*y(2);
        end
    end
end
```

Appendix B

MathWorks Inc. ballode.m example code representing event location dynamics [2].

```
(* ::Package:: *)

function ballode
tstart = 0;
tfinal = 20;
y0 = [0; 30];
refine = 4;
options = odeset ('Events',@events,'OutputFcn',
        @odeplot,'OutputSel',1,'Refine',refine);

figure;
set (gca,'xlim',[0  30],'ylim',[0  25]);
box on
hold on;

tout = tstart;
yout = y0.';
teout = [];
yeout = [];
ieout = [];
for i = 1:10
  % Solve until the first terminal event.
  [t,y,te,ye,ie] = ode23 (@f,[tstart  tfinal],y0,options);
  if ~ishold
    hold on
  end
  % Accumulate output.
```

```
  % This could be passed out as output arguments.
  nt = length (t);
  tout = [tout; t (2:nt)];
  yout = [yout; y (2:nt,:)];
  teout = [teout; te];  % Events at tstart are not reported.
  yeout = [yeout; ye];
  ieout = [ieout; ie];

  ud = get (gcf,'UserData');
  if ud.stop
    break;
  end

  % Set the new initial conditions, with .9 attenuation.
  y0(1) = 0;
  y0 (2) = -.9*y (nt,2);

  % A good guess of a valid first timestep is the length of
  % the last valid timestep, so use it for faster computation.
  % 'refine' is 4 by default.
  options = odeset (options,'InitialStep',t (nt)-t (nt-refine),
                            'MaxStep',t (nt)-t (1));

  tstart = t (nt);
end

plot (teout,yeout (:,1),'ro')
xlabel ('time');
ylabel ('height');
title ('Ball trajectory and the events');
hold off
```

```
odeplot ([],[],'done');

function dydt = f (t,y)
dydt = [y (2); -9.8];

function [value,isterminal,direction] = events (t,y)
% Locate the time when height passes through zero in
% a decreasing direction and stop integration.
value = y (1);      % detect height = 0
isterminal = 1;   % stop the integration
direction = -1;   % negative direction
```

Appendix C

Appendix C.1

MatLab Driverscript: Original drillstring program.

```
% MATLAB script file to simulate a drillstring
% ODE45 is used to simulate the drillstring,
% and results are plotted.

% Initialise parameters
Jr = 2122;       % Rotary Inertia, kg m^2
Jb = 471.9698;   % Bit inertia, kg m^2
Rb = 0.155575;   % Radius of the bit, m
ct = 172.3067;   % Torsional damping, N m s / rad
cr = 425;        % Damping on rotary inertia, N m s / rad
mucb = 0.5;      % Coulomb friction coefficient
musb = 0.8;      % Static friction coefficient
Dv = 0.000001;   % Size of transition region
gamb = 0.9;      % Speed of exponential decay
vf = 1;          %   "       "       "      "

%Parameters that most affect the long term behavior
kt = 861.5336;   % Torsional stiffness, Nm/rad
cb = 50;         % Damping on bit inertia, Nms/rad
u = 35000;        % Motor torque, N m
Wob = 29000;     % Weight on bit, N
tspan = [0, 100];
x0 = [0 0 0];
[t,x] = ode45(@ss_diff,tspan,x0,[],u,ct,cr,cb,kt,Jr, ...
        ...Jb, Rb, mucb,musb,Dv,gamb,vf,Wob);

subplot(1,2,1)
```

```
plot(t,x(:,1),'-r',t,x(:,3),'-k')
title('Inertia velocities')
legend('Rotary inertia','Bit inertia','Location','Best')
axis auto
xlabel('Time')
ylabel('Rotary Velocity')

subplot(1,2,2)
plot3(x(:,2),x(:,1),x(:,3))
xlabel('phi_r - phi_b (rad)')
ylabel('phidot_r')
zlabel('phidot_b')
grid on
```

Appendix C.2

MatLab Function Handle: Drillstring ode45 dynamics function file.

```
function dx = ss_diff(t,x,u,ct,cr,cb,kt,Jr,Jb,Rb, ...
                              ... mucb,musb,Dv,gamb,vf,Wob)
    tor = tfb;
    dx = zeros(3,1);
    dx(1) = (u-(ct+cr)*x(1)-kt*x(2)+ct*x(3))/Jr;
    dx(2) = x(1)-x(3);
    dx(3) = (ct*x(1)+kt*x(2) - (ct+cb)*x(3)-tor)/Jb;

    function tor = tfb
        Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
        Tsb = Rb*Wob*musb;
        if (abs(x(3))>=Dv)
                tor = Rb*Wob*(mucb+(musb-mucb)* ...
      ... exp(-gamb/vf*abs(x(3))))*sign(x(3)); %sliding
        else if (abs(Teb)>Tsb)
            tor = Tsb*sign(Teb); %stick-slip transition
            else
                tor = Teb; %stick
            end
        end
    end
end
```

Appendix C.3

MatLab Function Handle: Drillstring ode45 dynamics function file making use of the Taylor Series approximation to exponential function.

```
function dx = ss_diff_taylorseries(t,x,u,ct,cr,cb,kt, ...
                    ... ,Jr ,Jb,Rb,mucb,musb,Dv,gamb,vf,Wob)
    Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
    tor = tfb;
    dx = zeros(3,1);
    dx(1) = (u-(ct+cr)*x(1)-kt*x(2)+ct*x(3))/Jr;
    dx(2) = x(1)-x(3);
    dx(3) = (Teb - tor)/Jb;
    function tor = tfb
        Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
        Tsb = Rb*Wob*musb;
        if (abs(x(3))>=Dv)
 tor = Rb*Wob*(mucb+(musb-mucb)*( 1 + ...
(-gamb/vf*abs(x(3)))+(1/2)*(-gamb/vf*abs(x(3)))^2+ ...
(1/6)*(-gamb/vf*abs(x(3)))^3 + (1/24)...
*(-gamb/vf*abs(x(3)))^4 ))*sign(x(3));

        else if (abs(Teb)>Tsb)
                tor = Tsb*sign(Teb);
            else
                tor = Teb;
            end
        end
        end
end
```

Appendix C.4

MatLab Function Handle: Drillstring ode45 dynamics function file making use of a quotient-type decaying friction torque.

```
function dx = ss_diff_model8(t,x,u,ct,cr,cb,kt,...
    ... Jr,Jb,Rb,Rb,mucb,musb,Dv,gamb,vf,Wob)
    Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
    tor = tfb;
    dx = zeros(3,1);
    dx(1) = (u-(ct+cr)*x(1)-kt*x(2)+ct*x(3))/Jr;
    dx(2) = x(1)-x(3);
    dx(3) = (Teb - tor)/Jb;

    function tor = tfb
        Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
        Tsb = Rb*Wob*musb;

        if (abs(x(3)) < Dv)
            tor = min(abs(Teb), Tsb)*sign(Teb);
        else
            tor = Wob*Rb*(((musb-mucb)/
                (1+gamb*abs(x(3))))+mucb);
        end
    end
end
```

Appendix C.5

MatLab Driverscript: Drillstring program using prototype drillstring parameters.

```
% MATLAB script file to simulate a drillstring
% ODE45 is used to simulate the drillstring,
% and results are plotted.

% Initialise parameters
Jr = 0.518;       % Rotary Inertia, kg m^2
Jb = 0.0318;  % Bit inertia, kg m^2
Rb = 0.1;  % Radius of the bit, m
ct = 0.0001;  % Torsional damping, Nms/rad
cr = 0.18;        % Damping on rotary inertia, Nms/rad
mucb = 0.5;      % Coulomb friction coefficient
musb = 0.8;      % Static friction coefficient
Dv = 0.000001;  % Size of transition region
gamb = 0.9;      % Speed of exponential decay
vf = 1;          %   "        "       "      "

%Parameters that most affect the long term behavior
kt = 0.073;  % Torsional stiffness, N m / rad
cb = 0.03;         % Damping on bit inertia, Nms/rad
u =7.3;         % Motor torque, N m
Wob = 100;     % Weight on bit, N
tspan = [0,100];
x0 = [0 0 0];
[t,x] = ode45(@ss_diff_model5,tspan,x0,[],u,ct,cr,cb,...
  ...kt,Jr,Jb,Rb,mucb,musb,Dv,gamb,vf,Wob);
subplot(1,2,1)
plot(t,x(:,1),'-r',t,x(:,3),'-b')
title('Inertia velocities')
```

```
legend('Rotary inertia','Bit inertia','Location','Best')
axis auto
xlabel('Time')
ylabel('Rotary Velocity')

subplot(1,2,2)
plot3(x(:,2),x(:,1),x(:,3))
xlabel('phi_r - phi_b (rad)')
ylabel('phidot_r')
zlabel('phidot_b')
grid on
```

Appendix C.6

MatLab Function Handle: Drillstring ode45 dynamics function file making use of a combination of dry friction, switch and Karnopp's friction model.

```
function dx = ss_diff_model5(t,x,u,ct,cr,cb,kt,Jr,...
                    ...Jb,Rb,mucb,musb,Dv,gamb,vf,Wob)
    Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
    tor = tfb;
    dx = zeros(3,1);
    dx(1) = (u-(ct+cr)*x(1)-kt*x(2)+ct*x(3))/Jr;
    dx(2) = x(1)-x(3);
    dx(3) = (Teb - tor)/Jb;

    function tor = tfb
        Teb = ct*(x(1)-x(3))+kt*x(2)-cb*x(3);
        Tsb = Rb*Wob*musb;
        Tcb = Rb*Wob*mucb;

        if abs(x(3)) > Dv
            tor = Tcb*sign(x(3));    %sliding
        else if (abs(Teb)>Tsb)
        tor = Tsb*sign(Teb);  %stick-slip transition
            else
            tor = Teb;              %stuck
            end
        end
    end
end
```

Appendix D

Appendix D.1

KeYmaera bouncing ball example program

```
/**
 * Non-parametric hybrid bouncing ball example
 * [Sankaranarayananan, Sipma, Manna, HSCC'04].
 * h = height
 * v = velocity
 * H = height limit
 * g = gravitation
 * c = elastic dampening factor at floor (h=0)
 * provable
 */
\problem {
  \[ R h,v,t; R c,g,H,V\] (
    h=0 &v=16
  ->
    \[t:=0;
    (
      {h'=v , v'=-10 , t'=1 , h>=0};
      if (t>0 & h=0) then
        v := -v/2; t:=0
      fi
    )*@invariant(h=5*t^2+v*t & h>=0 &
        t>=0 & v<=-10*t+16 & t<=16/5)
  \] (0<=h & h<=13)
  ) }
```

Appendix D.2

KeYmaera water-tank example program

```
\functions {
}

/*
invariant:
y >=1 & y <=12 & (st=3 -> (y >= 5 - 2*x)) & (st=1->(y<=10+x))
*/

\problem {
\[ R x; R y; R st; x:=0; y:=1; st:=0 \] ( (st = 0)
->
 \[
(
  (?(st=0);
      (?(y = 10); x:=0; st:=1)
       ++ (?(y < 10 | y > 10); {x'=1,y'=1, y<=10})
  )
++
  (?(st=1);
       (?(x=2); st:=2)
        ++ (?(x < 2 | x > 2); {x'=1,y'=1, x <=2})
  )
++ (?(st=2);
        (?(y=5); x:=0; st:=3)
     ++ (?(y>5 | y < 5); {x'=1, y'=-2, y >=5})
   )
++ (?(st=3);
        (?(x=2); st:=0)
     ++ (?(x>2 | x < 2); {x'=1,y'=-2, x <= 2})
```

```
    )
)*@invariant(y >=1 & y <=12 & (st=3 -> (y >= 5 - 2*x)) &
                    (st=1->(y<=10+x)))
\]
(y >= 1 & y <= 12))
}
```

Appendix E

Appendix E.1

Original KeYmaera drillstring program.

```
\functions {
R Ueq(R,R,R);
R Teb(R,R,R);
}

\problem {
\[R x1, x2, x3, st;
        x1:=0 x2:=0; x3:=0 st:=3;

  R Jr, Jb, cr, cb, ct, kt, u, musb, mucb, Rb, Wob, delta, gamb, Vf, Tsb, Tcb, Ueq, Teb;
  Jr:=2122; Jb:=471.9; cr:=425; cb:=50; ct:=172.3; kt:=861; musb:=0.8;
        mucb:=0.5; Rb:=0.1555; delta:=0.0000001; gamb:=0.9; Vf:=1;

        u:=4300;
        Wob:=35500;
```

```
        Teb:=(ct*(x1-x3)+kt*x2-cb*x3);
        Ueq:=(ct*x1+kt*x2-ct*x3-cb*x3);
        Tsb:=Wob*Rb*musb;          Tcb:=Wob*Rb*mucb;
        \] (st=3
->
 \[
  ( ?(st=3);
      (?(x3 = 0 & Ueq(x1,x2,x3) > Tsb); x1:=x1; x2:=x2; x3:=x3; st:=1)
    ++(?(x3 = 0 & Ueq(x1,x2,x3) < -Tsb); x1:=x1; x2:=x2; x3:=x3; st:=2)
      ++(?(x3 = 0 & Ueq(x1,x2,x3) >= -Tsb & Ueq(x1,x2,x3) <= Tsb); {x1'=(-(ct+cr)*x1 -
                        kt*x2+u)/Jr, x2'=x1, x3'=0, x3=0}; st:=3)
  )

++(?(st=2);
      (?(x3 = 0 & Ueq(x1,x2,x3) > Tsb); x1:=x1; x2:=x2; x3:=x3; st:=1)
    ++(?(x3 = 0 & Ueq(x1,x2,x3) >= -Tsb & Ueq(x1,x2,x3) <= Tsb); x3:=0; st:=3)
      ++(?(x3 < 0 | (x3=0 & Ueq(x1,x2,x3) < -Tsb )); {x1'=(-(ct+cr)*x1-kt*x2+ct*x3+u)/Jr,
      x2'=x1-x3, x3'=(Teb(x1,x2,x3)+(Tcb+(Tsb-Tcb)*exp((-gamb/Vf)*-x3)) )/Jb, x3 < 0 };
                        st:=2)
  )
```

```
++(?(st=1);
          (?(x3 = 0 & Ueq(x1,x2,x3) < -Tsb);   x1:=x1; x2:=x2; x3:=x3; st:=2)
        ++(?(x3 = 0 & Ueq(x1,x2,x3) >= -Tsb & Ueq(x1,x2,x3) <= Tsb); x3:=0; st:=3)
          ++(?(x3 > 0 | (x3=0 & Ueq(x1,x2,x3) > Tsb)); {x1'=(-(ct+cr)*x1-kt*x2+ct*x3+u)/Jr,
          x2'=x1-x3, x3'=(Teb(x1,x2,x3)-(Tcb+(Tsb-Tcb)*exp((-gamb/Vf)*x3)) )/Jb, x3 > 0 };
                              st:=1)
   )
\] (x3 > 0 | (x3=0 & Ueq(x1,x2,x3) > Tsb))
)
}
```

Appendix E.2

Five location automaton representation of the drillstring with a combination of dry friction, switch and Karnopp's friction models presented on the prototype drillstring parameter set.

```
\functions {
R Ueq(R,R,R);
R Teb(R,R,R);
}

\problem {
\[ R x1, x2, x3, st, Tsb, Tcb, Ueq, Teb;
   R Jr, Jb, cr, cb, ct, kt, u, musb, mucb, Rb, Wob, Dv, gamb, Vf;

        Jr:=0.518; Jb:=0.0318; cr:=0.18; cb:=0.03; ct:=0.0001; kt:=0.073; musb:=0.8;
        mucb:=0.5; Rb:=0.1; Dv:=0.0000001; gamb:=0.9; Vf:=1;

         Wob:=100;
/* u:=8.3;                    positive velocity   */
   u:= 7.3;           /* Stuck     */
/* u:= 7.4;       Stick-slip   */
```

```
            Teb:=(ct*(x1-x3)+kt*x2-cb*x3);
            Ueq:=(ct*x1-ct*x3+kt*x2-cb*x3);
            Tsb:=Wob*Rb*musb;
            Tcb:=Wob*Rb*mucb;

            x1:=0.1; x2:=10; x3:=0.1; st:=1\] ((st=1)
->
 \[
  (?(st=1);
        (?(x3 > Dv); {x1'=(-(ct+cr)*x1-kt*x2+ct*x3+u)/Jr, x2'=x1-x3, x3'=(Teb(x1,x2,x3)
        - (Tcb) )/Jb, x3 > 0 }; st:=1)
        ++(?(x3 < Dv);  x1:=x1; x2:=x2; x3:=x3; st:=2)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) > Tsb)); x3:=0; st:=3)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) < -Tsb)); x3:=0; st:=4)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) <= Tsb & (Ueq(x1,x2,x3) >=
        -Tsb))); x3:=0; st:=5) )

++(?(st=2);
        (?(x3 > Dv); x1:=x1; x2:=x2; x3:=x3; st:=1)
        ++(?(x3 < Dv); {x1'=(-(ct+cr)*x1-kt*x2+ct*x3+u)/Jr, x2'=x1-x3, x3'=(Teb(x1,x2,x3)
```

```
        - (-Tcb) )/Jb, x3 < 0 }; st:=2)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) > Tsb)); x3:=0; st:=3)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) < -Tsb)); x3:=0; st:=4)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) <= Tsb & (Ueq(x1,x2,x3) >= -Tsb)));
        x3:=0; st:=5) )

++(?(st=3);
        (?(x3 > Dv); x1:=x1; x2:=x2; x3:=x3; st:=1)
        ++(?(x3 < Dv); x1:=x1; x2:=x2; x3:=x3; st:=2)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) > Tsb));{x1'=(-(ct+cr)*x1-kt*x2+ct*x3+u)
        /Jr, x2'=x1-x3, x3'=(Teb(x1,x2,x3)-(Tsb))/Jb, x3=0}; x3:=0; st:=3)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) < -Tsb)); x3:=0; st:=4)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) <= Tsb & (Ueq(x1,x2,x3) >= -Tsb)));
        x3:=0; st:=5) )

++(?(st=4);
        (?(x3 > Dv); x1:=x1; x2:=x2; x3:=x3; st:=1)
        ++(?(x3 < Dv); x1:=x1; x2:=x2; x3:=x3; st:=2)
      ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) > Tsb)); x3:=0; st:=3)
      ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) < -Tsb));{x1'=(-(ct+cr)*x1 -
```

```
        kt*x2+ct*x3+u)
       /Jr, x2'=x1-x3, x3'=(Teb(x1,x2,x3)+(Tsb))/Jb, x3=0}; x3:=0; st:=4)
       ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) <= Tsb & (Ueq(x1,x2,x3) >= -Tsb)));
       x3:=0; st:=5) )

++(?(st=5);
        (?(x3 > Dv); x1:=x1; x2:=x2; x3:=x3; st:=1)
        ++(?(x3 < Dv); x1:=x1; x2:=x2; x3:=x3; st:=2)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) > Tsb)); x3:=0; st:=3)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) < -Tsb)); x3:=0; st:=4)
        ++(?((x3 < Dv & x3 > -Dv) & (Ueq(x1,x2,x3) <= Tsb & (Ueq(x1,x2,x3) >=
        -Tsb)));{x1'=(-(ct+cr)*x1-kt*x2+u)/Jr, x2'=x1, x3'=0, x3=0}; x3:=0; st:=5) )

\] (x3 > 0 | (x3=0 & Ueq(x1,x2,x3) > Tsb))
)
}
```

Appendix E.3

Three location representation of the drillstring system with exponential-type friction model, presented with no functions.

```
\problem {
\[R x1, x2, x3, st;
 x1:=0.1; x2:=10; x3:=0.1; st:=1;

  R Jr, Jb, cr, cb, ct, kt, u, musb, mucb, Rb, Wob, delta, gamb, Vf;
  Jr:=2122; Jb:=471.9; cr:=425; cb:=50; ct:=172.3; kt:=861; musb:=0.8;
  mucb:=0.5; Rb:=0.1555; delta:=0.0000001; gamb:=0.9; Vf:=1;

  u:=3500;
  Wob:=70000;

        \] (st=1
->
 \[
  (
  ( ?(st=3);
        (?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) > Wob*Rb*musb); x1:=x1; x2:=x2; x3:=x3; st:=1)
```

```
        ++(?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) < -Wob*Rb*musb); x1:=x1; x2:=x2; x3:=x3;
         st:=2)
         ++(?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) >= -Wob*Rb*musb & (ct*x1+kt*x2-ct*x3-cb*x3)
          <= Wob*Rb*musb);{x1'=(-(ct+cr)*x1-kt*x2+u)/Jr, x2'=x1, x3'=0, x3=0}; st:=3) )

++(?(st=2);
        (?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) > Wob*Rb*musb); x1:=x1; x2:=x2; x3:=x3; st:=1)
        ++(?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) >= -Wob*Rb*musb & (ct*x1+kt*x2-ct*x3-cb*x3)
         <= Wob*Rb*musb); x3:=0; st:=3)
         ++(?(x3 < 0 | (x3=0 & (ct*x1+kt*x2-ct*x3-cb*x3) < -Wob*Rb*musb )); {x1'=(-(ct+cr)*
         x1 - kt*x2+ct*x3+u)/Jr, x2'=x1-x3, x3'=((ct*(x1-x3)+kt*x2-cb*x3)+(Wob*Rb*mucb+
         (Wob*Rb*musb - Wob*Rb*mucb)*exp((-gamb/Vf)*-x3)) )/Jb, x3 < 0 }; st:=2) )

++(?(st=1);
        (?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) < -Wob*Rb*musb);  x1:=x1; x2:=x2; x3:=x3; st:=2)
        ++(?(x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) >= -Wob*Rb*musb & (ct*x1+kt*x2-ct*x3-cb*x3)
        <= Wob*Rb*musb); x3:=0; st:=3)
        ++(?(x3 > 0 | (x3=0 & (ct*x1+kt*x2-ct*x3-cb*x3) > Wob*Rb*musb)); {x1'=(-(ct+cr)*x1 -
        kt*x2+ct*x3+u)/Jr, x2'=x1-x3, x3'=((ct*(x1-x3)+kt*x2-cb*x3)-(Wob*Rb*mucb+
        (Wob*Rb*musb - Wob*Rb*mucb)*exp((-gamb/Vf)*x3)) )/Jb, x3 > 0 }; st:=1) )
```

```
   )
\] (x3 > 0 | x3 = 0 & (ct*x1+kt*x2-ct*x3-cb*x3) > Wob*Rb*musb )
)
}
```