INDUSTRIAL APPLICATION OF THE SYSTEM-LEVEL SIMPLEX ARCHITECTURE FOR REAL-TIME EMBEDDED SYSTEM SAFETY

BY

STANLEY ZBIGNIEW BAK

B.S., Rensselaer Polytechnic Institute, 2007

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Associate Professor Marco Caccamo
Abstract

Cyberphysical systems differ from traditional computer programming applications in that software bugs and mistakes have immediate, possibly dangerous, physical-world consequences. Rather than requiring all control software be formally verified, which is often impractical, we develop a framework based on the Simplex Architecture, where an untrusted complex controller is allowed to actuate the plant while being monitored by a simpler, verified safety controller and associated decision logic. We use hardware/software co-design to eliminate large classes of potential software bugs associated with system control by developing the System-Level Simplex Architecture. Then, within an industrial case study, we use model-checking to help create and verify the safety-critical decision logic component of the System-Level Simplex Architecture, in the end generating verified VHDL (hardware) code.
Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Marco Caccamo, who guided me throughout the research and thesis process. My other PhD adviser Lui Sha was essential to guiding the development of the System-Level Simplex Architecture. Ashley Greer and the folks at John Deere also significantly helped shape this thesis. Thanks to the University of Illinois at Urbana-Champaign Computer Science department for twice awarding me the Debra and Ira Cohen Graduate Fellowship (and thanks to Debra and Ira Cohen for sponsoring the fellowship), which helped provide financial means to perform this work. Finally, thanks to parents and friends who helped relieve the stress inherent in performing research.

Although industrial support was essential to the completion of this thesis, any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of John Deere.
# Table of Contents

Chapter 1 Introduction ......................................................... 1
  1.1 Background Knowledge .................................................. 1
      1.1.1 The Simplex Architecture ......................................... 2
      1.1.2 Other Related Work ............................................... 3
  1.2 Thesis Overview ............................................................ 4

Chapter 2 System-Level Simplex ............................................. 5
  2.1 System-Level Versus Application-Level ................................ 5
  2.2 Inverted Pendulum Prototype ........................................... 6
  2.3 Cardiac Pacemaker Prototype ........................................... 8

Chapter 3 Application Within Industry ..................................... 11
  3.1 Application Overview .................................................. 11
  3.2 Formal Problem Description ........................................... 13
  3.3 Discrete Problem Approach ............................................ 16
  3.4 Model Generation ....................................................... 17
  3.5 Properties and Behavior ............................................... 20
      3.5.1 Rollover .......................................................... 20
      3.5.2 Overslip ......................................................... 22
      3.5.3 Oversteer ....................................................... 23
      3.5.4 Collision ....................................................... 24
  3.6 Behavior Determination ................................................. 24
      3.6.1 Rollover .......................................................... 25
      3.6.2 Overslip ......................................................... 26
      3.6.3 Oversteer ....................................................... 28
  3.7 Behavior Description and VMaude ..................................... 29
      3.7.1 Maude Decision Module Behavior Description ..................... 29
      3.7.2 VMaude Decision Module Behavior Description .................. 31
  3.8 Code Generation .......................................................... 35
  3.9 Improving Model-Checking Run Time ................................... 36

Chapter 4 Conclusions ....................................................... 39

Appendix A Code Reference .................................................. 40
  A.1 Inverted Pendulum VHDL Code ......................................... 40
  A.2 Model Generator Code .................................................. 47
  A.3 Sample Model File ..................................................... 62
  A.4 VMaude Semantics ...................................................... 64
  A.5 Roll VMaude Code ...................................................... 68
  A.6 Roll VHDL Code ........................................................ 69
  A.7 VHDL Generator Code ................................................... 72

References ................................................................. 78
Chapter 1

Introduction

Modern cyberphysical systems are growing in complexity and must meet strict requirements involving reliability, safety, and performance. Reliability and safety are difficult to scale using traditional designs because large systems have high complexity, and high system complexity creates more possibilities for errors. Additionally, high performance often demands significant system complexity, intensifying the problem.

One approach which deals with complex systems in safety-critical environments is the Simplex Architecture [7,19–21]. This architecture provides safety guarantees by “using simplicity to control complexity”. It uses a simple safety controller subsystem to ensure the stability of the plant. This conservative safety control core is then complemented by a high-performance complex control subsystem. A decision module then uses the high-performance complex controller whenever possible, but will switch to the safety controller when safety is jeopardized. The Simplex design has been previously been applied to improve the safety of a diving controller [8], a fleet of remote-controlled cars [7], and a set of advanced aircraft maneuvers [18].

There are two major concerns when the Simplex Architecture is used in a safety-critical system, both of which are addressed in this thesis.

- First, using Simplex as originally described, at the software level, creates a large body of unverified dependencies. For example, if the simple safety controller and the high-performance complex controller both use a shared operating system, a bug in the operating system may destabilize the plant. By using hardware/software co-design, the System-Level Simplex Architecture eliminates such common dependencies and therefore provides more robust control.

- Second, applying Simplex to a system requires a verifiably correct safety controller and decision module, which are nontrivial to produce. Previous work [20] used control theory to produce these subsystems in a continuous state space. However, no techniques has previously been provided which can verifiably use Simplex in discrete or hybrid (mixed discrete and continuous) systems. We go through an in-depth case study which uses model-checking to construct a verifiably correct trusted computing base for the System-Level Simplex Architecture. The case study focuses on control of autonomous John Deere vehicles, and the investigated models are not always continuous.

1.1 Background Knowledge

In order to put the thesis into context, we begin by providing background knowledge about the thesis topic. First, an overview of previously-developed Simplex research is provided in Section
1.1.1. Then, other related ideas are given in Section 1.1.2.

1.1.1 The Simplex Architecture

The original Simplex Architecture provides a fail-operational mechanism for a malfunctioning software controller. The architecture permits online modification and upgrade to control software without sacrificing safety. It is a robust design where complex controller bugs can be detected and fixed during run-time without jeopardizing safety.

The Simplex Architecture uses three subsystems: safety, complex, and decision. The safety subsystem has a simple, reliable controller which provides verifiably safe performance. This is used in case the complex controller malfunctions. The complex subsystem drives the system during regular operation, as long as it does not jeopardize system liveliness. This controller can be changed and upgraded while the system is running and may even contain bugs (therefore it does not need to be verified, or may even be too complex to fully verify). The decision subsystem chooses which of the two previously-mentioned controllers to use. An overview of the system is shown in Figure 1.1.

![Simplex Architecture Diagram](Image)

Figure 1.1: The Simplex Architecture maintains safety in spite of errors by switching between two controllers.

The safety subsystem and decision subsystem must be verified to guarantee that, in every possible way the complex controller can malfunction, the safety controller will actuate the system in time to prevent a violation of the operational constraints. In a continuous space, one way to construct these two components correctly leverages on control theory.

First, a set of operational state constraints are obtained to provide the definition of system liveliness. Next, we can obtain the safe region by calculating a Lyapunov function [3]. The safety controller should always be able to prevent violations of operational constraints if used within the safe region. We then take a subset of the safety state region, the recoverable region, to be the region where we use the complex controller, such that leaving the smaller state space is interpreted as being in danger of committing a safety violation. If the system leaves the recoverable region (perhaps because of a complex controller malfunction), the safety controller takes control and drives the system, thus preserving system liveliness. Of course, the system designer must ensure the safety controller has a chance to take over while still in the safe region; our state should not be able to
jump from inside the recoverable region to outside the safe region in a single control iteration. An abstract representation of these state spaces in two dimensions is shown in Figure 1.2.

![Figure 1.2: The safe region is a set of states with which the safety controller can maintain system liveliness. This can be determined in a continuous state space through a Lyapunov function. The recoverable region is a subset of the safe region that can tolerate aggressive actuation without exiting the safe region in a single control iteration.](image)

We stress that a good decision module will use the complex controller whenever it is functioning correctly and only switch to the safety controller when in danger of violating the safety condition.

1.1.2 Other Related Work

Other research has been performed on producing robust and reliable cyberphysical systems. One method such method is N-version programming [4]. In this method, multiple versions of software are independently created from the same specification. Then, all are run and the result given by the majority of versions is taken as the output of the system. One drawback with this method is the lack of statistical independence of bugs [5,9]. Additionally, for a constant amount of development effort, N-version programming is actually less reliable than focusing on a single version over a wide range of parameter values [20].

Another reliability mechanism is the recovery block concept [15]. In this approach, several alternative methods are developed. We first run the fully featured one and check if it is correct. If it is, we use it. Otherwise, we try the simpler ones. The essential difference between recovery blocks and the Simplex Architecture is that the former is a backward recovery method while the latter is a forward recovery method.

A common engineering practice to increase system reliability in spite of unreliable hardware is triple modular redundancy (TMR) [11]. In this scheme, three versions of identical hardware running an identical program are run with the same input. The output is then voted upon, such that if any one of the outputs is incorrect (due to a hardware failure or random environmental interference [12]), the overall system continues to function correctly. This technique, unlike the Simplex Architecture, is powerless against errors in the logic of the program, since all three modules will produce the identically incorrect output. However, it is effective against random hardware failures and other transient faults and thus can be used in conjunction with the Simplex Architecture to prevent both logical and random errors.
1.2 Thesis Overview

This thesis is divided into two main chapters for the two main contributions. First, we describe the System-Level Simplex Architecture in Chapter 2, which uses hardware/software co-design to eliminate common dependencies between the Simplex subsystems. Next, in Chapter 3, we go through an in-depth case study which uses model checking to construct a System-Level Simplex system targeted for partially autonomous John Deere tractors. These two main chapters are followed up by key insights and conclusions in Chapter 4. Finally, in Appendix A, we include the source code for several important software programs used within the industrial case study.
Chapter 2

System-Level Simplex

The original Simplex design, which we will now refer to as the Application-Level Simplex Architecture, can still be vulnerable to certain types of faults, particularly bugs in the operating system, middleware, and the underlying microprocessor. We address these concerns in Section 2.1 by introducing the novel idea of the System-Level Simplex Architecture. Then, we demonstrate the robustness of System-Level Simplex Architecture by building two prototypes. In Section 2.2, we describe an inverted pendulum prototype, and in Section 2.3, we describe a prototype cardiac pacemaker built using the System-Level Simplex design.

2.1 System-Level Versus Application-Level

Simplex, as described earlier in Section 1.1.1, contains three main subsystems. The complex controller is an unverified component which drives the system under normal operation. If the complex controller misbehaves, the safety controller takes over prior to the system collapsing and recovers the plant. The decision module determines when to perform the switching action, so as to never violate a particular safety requirement.

The Application-Level Simplex Architecture has all three subsystems located at the application-level. This works well for protecting the system from faults directly from complex controller, however it does not provide safety for indirect faults of underlying components. For example, if the operating system which runs all three subsystems in the Application-Level Simplex Architecture contains a bug and crashes, plant safety can no longer be guaranteed. In fact, the microprocessor, the operating system, and any middleware are required to be correct for a verifiable safe system. We relax this requirement in the System-Level Simplex Architecture by performing hardware/software partitioning on the system. The two Simplex safety-critical components, the safety controller and the decision module, are moved into a dedicated processing unit outside of the microprocessor.

Notice that this is hardware/software co-design, except that we perform this move not for typical reasons of performance and power consumption, but instead to protect from software-related faults. This is a novel idea, using hardware/software co-design to improve system safety.

The system designer has a choice for the dedicated processing unit which will run the Simplex safety core. One option is to use a microcontroller to run the two safety core subsystems. However, in safety-critical systems, even processors are not completely trusted [24], and we would prefer to eliminate this underlying complexity (we will touch on this point in our pacemaker prototype in Section 2.3). We instead run the Simplex safety core on dedicated hardware. Ideally, for a production version, we would produce an Application-Specific Integrated Circuit (ASIC). Instead,
to save time and money, we opted to construct our prototypes using Field Programmable Gate Arrays (FPGAs). The same VHDL code used to program an FPGA can be used to produce an ASIC.

By moving the Simplex safety core to isolated hardware, we can also provide temporal correctness for the monitored safety properties. If the high-performance complex subsystem does not produce a control command in the appropriate time, whether caused by an RTOS bug, poor cache performance or excessive bus contention, the conservative safety controller’s output is used. Since the safety controller runs in parallel on isolated hardware (which prevents run-time variations caused by resource sharing), the temporal constraints are met by design.

We now describe two prototype systems which we developed using the System-Level Simplex Architecture. First, we describe an inverted pendulum, and later we provide details about our cardiac pacemaker prototype.

2.2 Inverted Pendulum Prototype

An inverted pendulum is a classical control testbed where a rod must be maintained upright by moving a cart along a track. Additionally, an inverted pendulum presents an obvious failure state when the rod falls over. We implemented the System-Level Simplex Architecture on an inverted pendulum and evaluated its robustness in the presence of various failures. The VHDL source code for our inverted pendulum prototype is listed in Appendix A.1.

Automatic control for an inverted pendulum is a well-studied stability problem. By taking measurements of our specific inverted pendulum, we were able to generate code for the safety controller using Matlab Simulink. The generated code was 86 lines of C code which consisted of a sequence of simple matrix operations. We translated this C code into a VHDL state machine, which performed the required mathematical operations to drive the safety controller. The resultant VHDL module was 880 lines long and used a well-tested, open-source floating point unit.

The stability of an inverted pendulum can be modeled with a Lyapunov function. We take a subset of this stable region to be the recoverable region where aggressive action can be tolerated. When the state is in this recoverable region, we use the voltage value from the complex controller to actuate the motor on the cart of the inverted pendulum. Otherwise, the safety controller’s result is used.

Our hardware implementation resides on an externally-powered Xilinx ML505 FPGA. The FPGA uses a PCIe port to communicate with the software portion of the Architecture. The software portion can run on any computer capable of using the PCIe bus, although our current driver is written for Linux. The operating system does not even need to be aware of real-time requirements, as the System-Level Simplex Architecture will safely handle any timing faults that occur. Our implementation, however, uses Linux/RK as the OS for the complex controller. Through memory-mapped I/O, the complex controller reads the most recent angle and track position and suggests a motor voltage to the hardware-based decision module.

Through the implementation, we successfully demonstrated that the software-based complex controller was able to actuate the inverted pendulum as long as it did not jeopardize safety. When

---

1 Alternatively, we could have used fixed-point arithmetic if the complexity of the FPU was undesired.
the pendulum’s state left the recoverable region, the safety controller took over and prevented the pendulum from collapsing. In this way, we verified that the system was able to tolerate the following faults:

- **No Output**: The complex controller sends no output. This can occur, for example, if it enters an infinite loop.

- **Maximum Voltage**: The complex controller sends a control command to the motor to run at the highest speed possible.

- **Wrong Way — Maximum Voltage**: A control command is sent to the motor to go as fast as possible in the direction that will most quickly destabilize the pendulum.

- **Time Degraded Control**: A control bug design by a senior control engineer from Honeywell which, over time, provides increasingly degraded control. A video of this bug is available [25].

- **Operating System Crash**: The operating system running the complex controller crashes.

- **Timing Faults**: The software-based complex controller runs below its required frequency. This can be because of a lack of real-time scheduling, or an over-utilized system.

- **Computer Reboot**: The computer running the complex controller is shut down and restarted.

We outline two of the faults in detail: the wrong way bug, and the computer reboot bug.

**Wrong Way — Maximum Voltage**

The pendulum should remain balanced even if the complex controller outputs a motor voltage that would normally destabilize the system. This test took an extreme case of this where we used a working inverted pendulum controller for a few seconds, and then output the maximum voltage in the direction opposite of that needed to stabilize the pendulum. The decision module detected that the system state was no longer in the recoverable region and switched control to the safety subsystem. The safety controller returned the pendulum to a stable state and control was again given to the complex subsystem after a few seconds. Measurements from one iteration of this process are given in Figure 2.1.

**Computer Reboot**

The System-Level Simplex Architecture provides protection from arbitrary operating system behavior, including rebooting the system. From the decision module’s perspective, the computer rebooting is equivalent to a complex controller that sends no output. We ran this test on our inverted pendulum setup, and the pendulum remained stable throughout the reboot process. Additionally, after the computer restarted, the software-based complex controller was able to regain control of the inverted pendulum using memory-mapped I/O with the FPGA. This is significant because a common remedy for software problems is rebooting the computer; a malfunctioning complex controller can be repaired in this fashion while the system remains stable and safe.
Figure 2.1: When the inverted pendulum state passes the edge of the recoverable region (dashed line), the safety controller takes over and prevents system collapse.

2.3 Cardiac Pacemaker Prototype

An artificial pacemaker is a piece of hardware inserted into a patient’s body in order to regulate his or her heart rate. Detailed designs of cardiac pacemakers have been disclosed [17, 23]. Here, we examine the practicality and usefulness of the System-Level Simplex Architecture in this real-time safety-critical application.

We investigate two practical considerations for using the System-Level Simplex Architecture:

- Can the system be divided up into a safe controller and a complex controller such that the most likely causes of failure are contained in the complex controller?
- Is implementing the safety controller and decision module on FPGA hardware possible and practical?

The first concern, the division of the system into complex and simple controllers, is domain-specific. In order to address this, we must examine some properties of artificial pacemakers.

The first generation of artificial pacemakers actuated the heart at a set interval. This functionality was sufficient to keep the patient alive, however, problems did arise. For example, when a healthy person walks up stairs or performs strenuous action, his heart rate increases. The first generation of pacemakers did not take this into account and patients would become dizzy or uncomfortable throughout the day. Additional functionality was added to pacemakers to detect if the heart rate should be increased by monitoring the patient’s body, for example using accelerometer readings in conjunction with advanced algorithms to infer the patient’s current activity level [23]. Requirements were then added on top of this to preserve smooth heart-rate transitions, rather than suddenly jumping from 65 to 120 beats per minute because of a sudden large acceleration. Additionally, modern pacemakers attempt to detect and log anomalous events with the heart to aid a doctor’s diagnosis. The logged data must be retrieved, and this is done through wireless communication with an external device.

Modern pacemakers have many other requirements, however we already covered enough to apply
the System-Level Simplex Architecture. The rate-adaptive pacing modes, where the heart rate changes over time, require complex functionality. The pacing rate to which we should change is a function of the current rate, as well as the past and present accelerometer readings. The safety properties we want to enforce are that the heart rate should be between a lower rate limit and an upper rate limit, and should not change by more than some rate smoothing parameter. These are the properties we codify into the decision module. The safety controller is a simple implementation that meets our requirements. We choose a safe controller that slows down the heart to the resting rate (lower rate limit) in a way that satisfies the rate smoothing requirement. This safety controller does not have rate-adaptive functionality, but instead it acts a fail-operational mode that will maintain safety for the patient.

Our second concern is the feasibility of implementing the Simplex safety core on dedicated hardware. Software engineering is typically much easier than programming an algorithm in hardware, so we must confirm that the safety controller and decision module are practical to implement. In our prototype, we take advantage of reprogrammable hardware components\(^2\) (FPGAs). In order to program an FPGA we must provide a description of the hardware using a Hardware Description Language (HDL) such as VHDL. In the case of the pacemaker, the pacing controller is described using a finite state machine.

The finite state machine description of the pacemaker behavior is also useful in verification. The pacemaker finite state machine model was manually described in Maude \([6]\) and exhaustively searched using the Maude model checker for safety violations, for all possible behaviors of the complex controller. The idea of using model-checking to produce a safe decision module is explored in great detail during the industrial case-study presented in Chapter 3.

Implementing the pacemaker on a modern FPGA also provides us with a means to control power consumption. Xilinx FPGAs have several clock regions which can be toggled on or off \([1]\). By implementing a soft processor on the FPGA for the complex subsystem, the processor’s clock can be disabled when the battery is low. In CMOS circuits, preventing transistor state changes (by stopping the clock) results in near-zero power consumption. This provides the System-Level Simplex the possibility to provide power isolation from the complex controller.

**Comparison with Existing Pacemaker Reliability Mechanisms**

Artificial pacemakers are safety-critical systems and as such they already contain some reliability mechanisms. We compare the existing reliability mechanisms found in one previous-generation pacemaker description \([23]\) to the proposed design created using the System-Level Simplex Architecture. We focus on two mechanisms for enhanced reliability which were present in the pacemaker description we examined.

The first is a watchdog timer which is periodically reset during normal system execution. If the execution hangs at some point, the timer will not be reset and will timeout. The timeout triggers a high-priority interrupt which signals that an anomalous event has occurred and the system is reinitialized. Alternately, the system can instead shut down as a fail-safe mechanism.

The watchdog timer mechanism is compatible with the System-Level Simplex Architecture. It provides a means to restart the system when it enters a rare error state. However, the watchdog

\(^2\)A production pacemaker would likely use an application-specific integrated circuit (ASIC) for the final product. ASICs can be manufactured using the same VHDL code used to program an FPGA.
timer does not protect the system from unsafe pacing, only system hangs. Additionally, deterministic bugs in the program will continue to restart the system, whereas a System-Level Simplex system is able to function safely in spite of deterministic bugs in the complex controller.

The other safety mechanism we examine is a redundant pacemaker system which, at first, appears to be similar to the System-Level Simplex Architecture. This system provides a simpler pacing mode without rate-adaption. This system is activated, and the microprocessor is deactivated when “a fault is detected in the operation of the microprocessor circuit.” This component, like the System-Level Simplex Architecture, provides protection from microprocessor errors. This is a real cause of concern with this specific pacemaker design because it uses a custom pacemaker-specific microprocessor. However, it does not provide protection from logical faults in the software. Additionally, control is switched to this system when any fault in the microprocessor is detected. In the System-Level Simplex Architecture, a microprocessor fault that only affects the logging mechanism (perhaps because of a rarely used instruction), one that does not compromise safety, would not mandate a switch to the safety controller.
Chapter 3

Application Within Industry

To provide further insight into the Simplex Architecture and its application, we now go through an in-depth case study applying the architecture towards control of partially autonomous tractors. While describing the case study, in addition to presenting our conclusions, we will also explain key insights we discovered while going through the process.

We will first provide an overview of the application in Section 3.1. Next, we describe the formal model generation process in Section 3.4. Then, we outline the three specific properties we investigated in Section 3.5, and the process of determining a verifiably correct decision module for each property in Section 3.6. Exactly describing the decision module behaviors is done in Section 3.7, followed up by VHDL-code generation in Section 3.8 and insights about improving model-checking run time in Section 3.9.

3.1 Application Overview

In collaboration with John Deere, we have been working to verify certain properties dealing with the behavior of autonomous tractors. We are not providing the autonomous behavior itself, but instead providing a way to guarantee that the tractor will never misbehave, in spite of a complex and unverified controller. This is particularly important for autonomous tractors, since a human operator brings a large amount of intuition that is used, but not immediately apparent when operating a tractor. For example, if a human is driving a tractor at a rapid velocity on a steep hill, he intuitively knows not to turn sharply uphill or the tractor may roll over. This may be left out of a high-level automated controller which is attempting to go from point A to point B as quickly as possible. Additionally, controllers are not perfect, so even if such considerations are present in the high-level code, their implementations may contain bugs or miss edge cases. Our goal then, is to be able to use a high-level controller which we did not develop, and be able to provide a guarantee that the composite system of the high-level controller and our Simplex instance always meets certain properties, for example never rolling over.

In contrast to the previous prototype implementations (the inverted pendulum and the rate-adaptive cardiac pacemaker), tractors are generally safe when they do nothing. For instance, to stop a tractor from rolling over it suffices to stop the tractor from moving. The challenge here is less in developing a fail-operational system, but more towards a fail-safe system. Thus, the majority of the development and verification effort is involved with developing the decision module, and the safety controllers are relatively simpler.

Given this goal of producing a verifiable decision module, we made a number of decisions up
front which impact the final system. First, we chose to use the System-Level version of the Simplex Architecture. Second, we deviated from the pure System-Level Simplex previously described for smoother system integration. Lastly, we used a discretized state space approximation of the continuous space the tractor exists in. We now describe and justify each of these decisions in detail.

As discussed earlier, the System-Level instantiation of the Simplex Architecture reduces the size of the trusted computing base in the system. In this application, we decided to use an FPGA as the platform for our Simplex Safety Core, the decision module and safety controller. If we had used the more traditional Simplex approach, bugs or failures in dependent modules such as the Real-Time Operating System (which for John Deere can be a commercial or internally-developed operating system) may result in misbehavior. This decision impacts the final code generation, as we targeted VHDL, a hardware description language.

In order to smooth integration with an existing Deere system, we deviated from the pure System-Level Simplex design we described in the earlier section. In a pure design, the complex controller would be on one side of the system, which was only connected to the Simplex module, which then connected to the rest of the system. This was the design used in the Inverted Pendulum testbed. However, to smooth integration we instead logically connected the system in this fashion, but physically all the communication went through a shared CAN bus. This was the least intrusive way to provide the System-Level Simplex functionality. This distinction is shown in Figure 3.1.

![Figure 3.1: The logical application design (left) matches the System-Level guidelines, although the physical design (right) differs. The dotted line represents actuation commands.](image)

Lastly, we used a discrete state space to aid in our verification step. The reason for discretizing the continuous state space was because we planned to use a model-checking tool, Maude [6], to verify the correctness of the decision logic. Model-checking tools work strictly on discrete state spaces, so we needed to perform this transformation. Notice that this is different from the technique used in previous Simplex publications [20]. Particularly, in earlier work, classical control theory was used directly within the continuous space to guide the decision module’s switching logic. Our approach works on the continuous space by mapping the continuous space into discrete elements. This has the advantage of being able to handle a larger class of scenarios. In particular hybrid systems, systems with both discrete and continuous elements, can be modeled with our model-checking approach, but
not with classical control theory. The disadvantage is that the mapping from a continuous space to
discrete elements must be done carefully. Informally, it can never be the case that an unsafe state in
the continuous space maps to a safe state in the discrete space. This is expressed formally in next
section.

3.2 Formal Problem Description

Now, we describe the class of problems we attempt to solve in a formal manner. Throughout the
description, we consider the toy problem of a train driving on a finite-sized straight track, which
should not fall off either end of the track (at position ± 20 relative to the start position of the
train). The train has four modes, forward (+2 m/s²), backward(−1 m/s²), cruising(0 m/s²),
and breaking(± 5 m/s² or 0 m/s²). Breaking mode always sets the acceleration against the current
velocity of the train (and is 0 if the train is still). Initially, the train is in cruising mode at position
0 with no velocity or acceleration.

Our model consists a state space consisting of n dimensions where each dimension has its own
domain D₁ to Dₙ. For example, in our train problem we consider four dimensions(n = 4), in
order, the acceleration, velocity, and the position of the train relative to start of the track, as well
as the mode the train is in. The ordering was done arbitrarily. The domains D₁ (acceleration
domain) = D₂ (velocity domain) = D₃ (position domain) = R. The domain of the mode, D₄ is
{forward, backward, cruising, breaking}.

A state of the system is an assignment of an element of the associated domain for each of the
dimensions. It is represented by an ordered tuple, σ = (σ₁, σ₂, ..., σₙ) which each σᵢ is an element of
the associated domain Dᵢ. For example, one possible state for our train system is (0, 2, 10.5, cruising),
corresponding to the state where the acceleration is 0, the velocity is 2, and the position is 10.5, and
the train is in cruising mode. The set of all possible states is of the system S.

Next, we define a time-transition function, τ : S × R → P(S) (where P denotes the powerset),
which describes how the state changes over time. The time-transition function takes as input a state
of the system, and a time duration, and gives us the set of possible resultant states of the system.
In an accurate deterministic system such as our train example, there is only one resultant state for a
given initial start and time duration. However, in other models, due to environmental disturbances
or actuator/sensor inaccuracies, we may get a set of possible states.

The τ function can not be arbitrary, but rather it must satisfy the Time-Transition Invariant,
which informally states that the resultant state set should only depend on the cumulative time that
has passed. This is formally defined in Equation 3.1.

$$\forall (x,y \in \mathbb{R}^+ , \sigma \in S) \tau(\sigma, x + y) = \bigcup_{\sigma' \in \tau(\sigma,x)} \tau(\sigma', y)$$ (3.1)

In our train example, we define a time-transition function which relates the position, velocity,
acceleration, and state of the train based on the mode of the train. Below, we define a case for each
scenario, forwards mode, backwards mode, cruising mode, breaking to a stop while moving forward,
breaking to a stop while moving backward, breaking while moving forward, breaking while moving
backward, and breaking while not moving. In particular,

$$\forall (accel, vel, pos \in \mathbb{R}, mode \in \{\text{forward, backward, cruising, breaking}\}, time \in \mathbb{R}^+),$$

\[\tau((accel, vel, pos, mode), time) =\]

\[
\begin{align*}
&\{(2, vel + 2 \times time, pos + vel \times time + time^2, forward)\} & \text{if mode} = \text{forward} \\
&\{(-1, vel - time, pos + vel \times time - \frac{1}{2} \times time^2, backward)\} & \text{if mode} = \text{backward} \\
&\{(0, vel, pos + vel \times time, cruising)\} & \text{if mode} = \text{cruising} \\
&\tau((accel, vel, pos, breaking), \frac{vel}{5}) & \text{if mode} = \text{breaking and vel} > 0 \\
&\text{and time} > \frac{vel}{5} \\
&\{(-5, vel - 5 \times time, pos + vel \times time - \frac{5}{2} \times time^2, breaking)\} & \text{if mode} = \text{breaking and vel} > 0 \\
&\text{and time} \leq \frac{vel}{5} \\
&\{(5, vel + 5 \times time, pos + vel \times time + \frac{5}{2} \times time^2, breaking)\} & \text{if mode} = \text{breaking and vel} < 0 \\
&\text{and time} \leq \frac{vel}{5} \\
&\{(0, 0, pos, breaking)\} & \text{if mode} = \text{breaking and vel} = 0
\end{align*}\]

The next step would be to show that this time-transition function satisfies the Time-Transition Invariant of Equation 3.1. Since the provided time-transition function \(\tau\) always results in one state and never modifies the \(mode\) dimension, we can decompose the proof for each mode. We now show the proof for when \(mode = \text{forward}\), although the other modes can also be proven in a similar fashion.

When \(mode = \text{forward}\), the time transition function is \(\tau((accel, vel, pos, forward), time) = \{(2, vel + 2 \times time, pos + vel \times time + time^2, forward)\}\). To prove the Time-Transition Invariant in this instance, we must show that:

$$\forall (x, y \in \mathbb{R}^+) \bigcup_{\sigma' \in \tau((accel, vel, pos, forward), x)} \tau(\sigma', y) = \tau((accel, vel, pos, forward), x + y)$$

The proof is direct. Consider any \(x, y \in \mathbb{R}^+\).}

$$\bigcup_{\sigma' \in \tau((accel, vel, pos, forward), x)} \tau(\sigma', y)$$

$$= \tau((2, vel + 2 \times x, pos + vel \times x + x^2, forward), y)$$

$$= (2, vel + 2 \times x + 2 \times y, pos + vel \times x + x^2 + (vel + 2 \times x) \times y + y^2, forward)$$

$$= (2, vel + 2 \times (x + y), pos + vel \times (x + y) + (x + y)^2, forward)$$

$$= \tau((accel, vel, pos, forward), x + y)$$

Given this definition of states and the time-transition function, each command is modeled as a state modification. This modification can not (usually) be done arbitrarily, but rather is subject to the problem-specific constraints.

For example, in our train system, commands correspond to changes in the \(mode\) dimension.
The are no actions corresponding to position changes directly, so modifying the position dimension during a command is not allowed.

The execution of the system can be thought of as a repeating cycle consisting of a command, which modifies the state directly, followed by the passage of time, which gives us a set of possible resultant states according to the time-transition function $\tau$.

Finally, within our formal model there is the notion of safe and unsafe states. This problem-specific notion divides the state space into states that are considered safe, and states which should be avoided and are considered unsafe. This notion of unsafe states is captured by the function $v : S \rightarrow \{\text{true}, \text{false}\}$. In our train example, we use this to define when the train has fallen off the track. Particularly,

$$v((\text{accel}, \text{vel}, \text{pos}, \text{mode})) = \begin{cases} 
\text{true} & \text{if } \text{pos} < 20 \text{ or } \text{pos} > 20 \\
\text{false} & \text{otherwise}
\end{cases}$$

We are now ready to formally specify the workings of the Simplex pattern within this architecture. In the systems we consider there is a control period, $c \in \mathbb{R}^+$, which determines the length of the control interval of the composite Simplex system. The complex controller, as well as the simple controller, run at this predetermined frequency, producing commands which are state modifications. We model the complex controller and simple controller, respectively, as functions $\alpha : S \rightarrow S$ and $\beta : S \rightarrow S$ which, given an input state $\sigma \in S$, produce an output state $\sigma' \in S$. These functions are not arbitrary, but rather can only modify dimensions of the system which the controllers are allowed to directly modify. For example, our train system controllers only have access to the mode dimension of the system.

If we are at the start of a control interval and the system is in state $\sigma \in S$, the complex controller and simple controller will output post-command states $\alpha(\sigma) = \sigma_{\text{complex}}$, and $\beta(\sigma) = \sigma_{\text{simple}}$. The decision module will then, given the current state of the system and the two proposed system-state modifications, choose one command and apply the corresponding system-state modification. This is also modeled as a function $\delta : S \times S \rightarrow S$. This is again, not an arbitrary function, but the output is restricted. Particularly, if $\delta(\sigma_{\text{complex}}, \sigma_{\text{simple}}) = \sigma_{\text{result}}$, then $\sigma_{\text{result}}$ must be equal to $\sigma_{\text{complex}}$ or $\sigma_{\text{simple}}$. After the command state modification, for the duration of the control interval $c$, the system will be modified naturally and the next state will be a state in the set of states obtained from the time-transition function $\tau$. At this point another command decision will be made repeating the procedure described above, although now with a different system state.

In general, the two main challenges involved with Simplex are determining the safety controller behavior and determining the decision logic behavior. However, for the example train system, as well as the John Deere tractor system we will soon describe, the safety controller is straightforward. It is clear, for example, that most direct way for the train to avoid falling off the edge of the track is to always set the mode to breaking\footnote{One may argue that it may be better to slow down prior to applying the breaks. However, the safety controller should be considered a backup that ideally would never be used. Therefore, we believe such comfort requirements are better suited as part of the complex controller.}. The main challenge, therefore, is to develop the decision module logic that will determine which command to apply. The primary requirement of the decision module is that the state of the system will never be unsafe. Formally the challenge is to come up
with a function $\delta$ that, for any function $\alpha$ (which is a valid control function in that it only modifies the controlled dimensions), satisfies Equation 3.2.

$$\forall (\sigma \in S) \neg \upsilon(\sigma) \Rightarrow \forall (t \in R^+ | t \leq s, \sigma' \in \tau(\delta(\alpha(\sigma), \beta(\sigma)), t)) \neg \upsilon(\sigma')$$ (3.2)

Here, $\beta$ is the known simple controller, and $s$ is the predetermined control period. The above equation states that for every safe state, the state during and at the end of the control period will also be a safe one. If the initial state is safe, then by induction the state will always be safe over all time.

There is another requirement for the decision module which is more informal. It requires that the complex controller’s commands should be used until we are in danger of entering an unsafe state. While the first requirement is related to safety-critical demands, this requirement deals with mission-critical aspects. For example, it is (probably) useless to use a decision module within our train example which always uses the safety controller (keeps the breaks on), over all time, even though the resultant system is safe and never falls off either end of the track.

### 3.3 Discrete Problem Approach

Given the problem described in the previous section, we now must determine a way to construct the appropriate decision logic function $\delta$. Our approach involves performing model checking, which is a proof technique which enumerates a finite set of states, checking the condition on each one. The condition we check is that the system state at the beginning of each control iteration is safe, and then recursively performing the same check for all states that result, for all possible commands from that state. As soon as a state is encountered which has already been checked, it can be dropped without rechecking.

Since we only check states at the start of each control iteration, this creates an additional requirement which must be satisfied. Particularly, if at any time during the time-elapsed portion of the control loop the system becomes unsafe, it should also be unsafe at the end of the time-elapsed portion of the control loop. Formally, the condition we need to show, for the particular model we are checking, is

$$\forall (\sigma \in S) \neg \upsilon(\sigma) \Rightarrow \forall (t \in R^+ | t \leq s) \neg \upsilon(\tau(\delta(\alpha(\sigma), \beta(\sigma)), t))$$

Additionally, in order for the model checking to complete, the number of safe states reachable from the initial state must be finite. This may not be true if the allowed commands at a safe state are infinite (for example, if the command sets a real-valued acceleration directly). Thus, in order to accommodate model-checking requirements, we discretize all continuous dimensions within our model. Formally, for each dimension in our model $i$, if the dimension is real-valued continuous, we must provide a discretization function, $\epsilon_i : R \rightarrow R$, which maps the continuous space to a finite set of values, as described in equation 3.3.
\[ \forall (x,y \in \mathbb{R} | x \leq y) \exists (n \in \mathbb{Z}) n > \left| \{ r \in \mathbb{R} | \exists (z \in \mathbb{R} | x \leq z \leq y) \epsilon_1(z) = r \} \right| \]  

(3.3)

This equation states that for any range \([x, y]\), the cardinality of the set resulting from discretizing all elements within the range is finite. This alone, however, does not guarantee there will be a finite number of states to search. In order to provide a finite search-state guarantee, we combine this discretization function with a bound on the allowed ranges on each real-valued dimension, which guarantees a finite number of possible values for each dimension, and therefore a finite number of states in the aggregate system. This is not as limiting as it appears. For example in our models is reasonable that a tractor will have a maximum and minimum velocity, or a maximum and minimum steering angle.

The discretization, however, must be done carefully, so that no unsafe states in the continuous domain get mapped to safe states in the discrete domain. If this is not the case, the result of the model-checking search is invalid and can not be trusted. This is made formal in Equation 3.4.

\[ \forall (i \in \mathbb{Z} | 0 < i \leq n) \text{ and } \mathcal{D}_i = \mathbb{R} \forall (\epsilon_1 \in \mathcal{D}_1, \epsilon_2 \in \mathcal{D}_2, \ldots, \epsilon_n \in \mathcal{D}_n) v((\epsilon_1, \epsilon_2, \ldots, \epsilon_n)) \Rightarrow v((\epsilon_1, \epsilon_2, \ldots, \epsilon_i(\epsilon_i), \ldots, \epsilon_n)) \]  

(3.4)

Again, \(n\) here represents the number of dimensions in the system. Although, we have only dealt with continuous dimensions that are real-valued, any ordered continuous dimension could follow similar requirements to guarantee a finite-set of reachable states whose model-checking will complete with a valid result.

The final concern is the time it takes to perform the model checking, which is very dependent on the discretization function \(\epsilon\) which is chosen. If each continuous dimension is discretized into millions of discrete values, exploring the aggregate system may become intractable. Thus, the choice of discretization function can be used to tune the performance of the model-checking search.

### 3.4 Model Generation

The first step in the industrial design process for System-Level Simplex is to create the model of the system for which we are interested. In our case we want to model movement-based properties of tractors. However, since John Deere produces several types of tractors, as well as combines and other vehicles, it would be prudent to have a common generator for all of these models. Additionally, by creating a model generator we are reducing the chance for human error while manually creating the model, as well as eliminating the need to know Maude (our model checking tool) before a model is constructed. The model generator source code is listed in Appendix A.2.

In order to decide which properties were important for the models, we first needed to figure out which specific properties we wanted to check. However, for sake of presentation we push that discussion into Section 3.5. We came up with a list of 27 model properties which are of interest, and we describe each of them here.

1. **TimeDiscretizationConstant**: The discrete unit of time, in milliseconds
2. **DistanceDiscretizationConstant**: The discrete unit of distance, in millimeters

3. **AngleDiscretizationConstant**: The discrete unit of angle measurement, in milliradians

4. **WheelBase**: The wheel base of the tractor, measured in millimeters

5. **Track**: The wheel base of the tractor, measured in millimeters

6. **CogHeight**: The height of the center of gravity, measured in millimeters

7. **ForwardsMinAccel**: The minimum acceleration when moving forward (breaking acceleration), in \( \text{millimeters} \cdot \text{second}^{-2} \).

8. **ForwardsMaxAccel**: The maximum acceleration when moving forward, in \( \text{millimeters} \cdot \text{second}^{-2} \).

9. **BackwardsMinAccel**: The minimum acceleration when moving backward, in \( \text{millimeters} \cdot \text{second}^{-2} \).

10. **BackwardsMaxAccel**: The maximum acceleration when moving backward (breaking acceleration), in \( \text{millimeters} \cdot \text{second}^{-2} \).

11. **MinVel**: The minimum velocity of the tractor, in \( \text{millimeters} \cdot \text{second}^{-1} \).

12. **MaxVel**: The maximum velocity of the tractor, in \( \text{millimeters} \cdot \text{second}^{-1} \).

13. **MinSteeringAngle**: The minimum steering angle of the tractor’s wheels, in milliradians. Straight wheels have a 0 milliradian measurement.

14. **MaxSteeringAngle**: The maximum steering angle of the tractor’s wheels, in milliradians. Straight wheels have a 0 milliradian measurement.

15. **DeltaSteeringAngle**: The speed at which the steering angle can change, in \( \text{milliradians} \cdot \text{second}^{-1} \).

16. **MaxSlope**: The maximum tilt of the terrain, in milliradians.

17. **MaxDeltaSlope**: The maximum change in slope at the maximum velocity of the tractor. This is expressed in \( \text{milliradians} \cdot \text{second}^{-1} \).

18. **BestRawAccel**: While on the least slippery surface, the acceleration of the wheels during the measurement. This is used for the overslip function, and is elaborated upon further in Section 3.5.

19. **BestObservedAccel**: While on the least slippery surface, the acceleration observed by the tractor during the measurement. This is used for the overslip function, and is elaborated upon further in Section 3.5.

20. **WorstRawAccel**: While on the most slippery surface, the acceleration of the wheels during the measurement. This is used for the overslip function, and is elaborated upon further in Section 3.5.

21. **WorstObservedAccel**: While on the most slippery surface, the acceleration observed by the tractor during the measurement. This is used for the overslip function, and is elaborated upon further in Section 3.5.
22. **TimeFromBestToWorst**: The time, in milliseconds that it takes the tractor to drive, at the maximum velocity, from the most slippery surface to the least slippery surface. Can be pessimistic.

23. **SkidBestRawAccel**: While on the least slippery surface, the expected acceleration of the turn based on the wheel angle during the measurement. This is used for the oversteer function, and is elaborated upon further in Section 3.5.

24. **SkidBestObservedAccel**: While on the least slippery surface, the acceleration observed of the turn during the measurement. This can be calculated using the radius of the turn and the time elapsed to complete the turn. This is used for the oversteer function, and is elaborated upon further in Section 3.5.

25. **SkidWorstRawAccel**: While on the most slippery surface, the expected acceleration of the turn based on the wheel angle during the measurement. This is used for the oversteer function, and is elaborated upon further in Section 3.5.

26. **SkidWorstObservedAccel**: While on the most slippery surface, the acceleration observed of the turn during the measurement. This can be calculated using the radius of the turn and the time elapsed to complete the turn. This is used for the oversteer function, and is elaborated upon further in Section 3.5.

27. **SkidTimeFromBestToWorst**: The maximum time, in milliseconds that it can take the tractor to drive from the least slippery surface to the most slippery surface. This can be the same value as **TimeFromBestToWorst**.

A sample file containing reasonable parameters has been given in our code reference in Appendix A.3. This file is used as input for a Java program which produces a three distinct tractor models, one for each of the properties we are checking. The advantage of having three different models instead of one common model, is that we are able to eliminate irrelevant dimensions within the model, and therefore decrease model-checking time. For example, the model dealing with the overslip property, which deals with how the wheels slip based on the surface, does not contain the steering dimension or the terrain slope dimension.

The model generator program itself, whose code is in our code reference Section 3.4, reads in the input file, and performs certain checks to make sure the generated model makes sense. Some of the checks are syntax-based, for example an ill-defined model file that contains multiple equals signs will not be accepted. Other checks are sanity checks, for instance if, during a skid measurement, the output acceleration, the acceleration of the tractor that was observed, is greater than the input acceleration, the acceleration of the tractor exerted by the wheels, an error is reported and the models are not generated. The last potential fatal error that can occur while running the generator is that the various dimensions do not fit together. For example, if the maximum speed of the tractor is \( \frac{1}{2} \text{m/s} \) and the distance discretization constant is \( 1\text{m} \), but the time discretization constant is \( 0.5\text{s} \), a model can not be generated. This is because if we are accelerating at full speed, after one time step, \( 0.5\text{s} \), the tractor is at a speed that does not match our distance discretization constant. In this case, the generator will report and error and exit. For different dimensions, for instance the friction-based dimensions, instead of exiting, the maximum slip is pessimistically decreased until the range of the
slip fits within the rest of the model. This pessimism is then reported as a warning so the user can chose to accept it, or rectify the model appropriately.

3.5 Properties and Behavior

Within an autonomous tractor system, we desire several properties to be monitored through the Simplex framework. Here, we present some of the properties we explored, three of which we applied Simplex towards, rollover, overslip, and oversteer, and one property that did not easily fit the Simplex framework, collision. Notice that these are not all safety-critical as is the traditional use for Simplex, but rather some properties such as overslip, the tractor slipping excessively, are properties that, if ignored, will cause in degraded performance (in the case of overslip, wasted fuel).

3.5.1 Rollover

Rollover is a condition that occurs when a tractor tips over, usually due to a sharp turn. Since tractors have an elevated center of gravity, rollover is a real concern. Prior to the widespread installation of tractor roll cages, under human operation, rollover accidents accounted for about 50% of tractor-related fatalities at a rate of 5.4 deaths per 100,000 tractors [10]. In the autonomous tractor scenario, the main concern is not rollover operator fatalities (since there is no operator), but rather damage to equipment, which for autonomous vehicles has an estimated cost from $10,000 to $200,000 [16]. Due to this cost, rollover should be avoided.

Within our system, we consider the tractor to be violating the rollover condition whenever one pair of wheels leaves the ground. Although according to the strict definition this is not yet rollover, such a condition should never come up during normal tractor operation and would thus like to be avoided. An outline of the rollover situation used for our modeling is shown in Figure 3.2. In the figure, \( \theta \) is the angle calculated by the provided track and center of gravity height. The other angle \( \alpha \) is the slope of the terrain which is equal to the roll angle of the tractor. \( F_{\text{lift}} \) is the lifting force due to the force of the turn, \( F_{\text{turn}} \), and \( F_{\text{stability}} \) is the stability force due to the force of gravity \( F_{\text{grav}} \). The velocity of the tractor is \( v \), the mass \( m \), and the angular velocity due to turning is \( \omega = v \sin(\theta)/\text{WHEEL-BASE} \).

Notice that there are two opposite forces, \( F_{\text{lift}} \) and \( F_{\text{stability}} \), which can be used to tell if tractor is rolling over. By solving for these forces, as shown in Figure 3.3, and comparing them, we can determine if the tractor is rolling over. The final rollover equation we use is

\[
g > v \omega \sin(\theta)/\cos(\theta + \alpha)
\]

where \( g \) is the force of gravity. Notice this equation make intuitive sense, as the best chance of violating the rollover condition occurs during a sharp turn uphill (this is the reason that turns on the on-ramps of highways or turns of race tracks are often banked so you are turning downhill).

In order to prevent rollover, the safety controller must reduce \( F_{\text{lift}} \). Our safety controller does this by reducing \( F_{\text{turn}} \), the contributing force to \( F_{\text{lift}} \), in two ways: 1) straightening the wheels, which reduces \( \omega \), and therefore \( F_{\text{turn}} \), and 2) slowing down, which reduces \( v \) and therefore also reduces \( F_{\text{turn}} \). Other options which were less practical were increasing the track distance or lowering the
Figure 3.2: The rollover condition is violated whenever $F_{\text{lift}}$ exceeds $F_{\text{stability}}$.

Figure 3.3: The values of $F_{\text{lift}}$ and $F_{\text{stability}}$ can be computed for comparison.
center of gravity (this would reduce \( \theta \) and therefore \( F_{\text{lift}} \)).

Reducing the velocity is almost always a valid action to take, however, there was some initial resistance to manipulating the steering angle. The complex controller may turning to avoid something, where it would be better for the tractor to rollover. However, when we determined the potential violation state region for rollover (described more in Section 3.6), if we were only manipulated the velocity, we noticed the tractor could not reach its maximum speed, even if the wheels were straight and the ground was flat. This was because if the complex controller turned the wheels as sharply as possible, and the terrain got more steep, the tractor could not slow down fast enough to avoid rollover. Therefore, to allow the complex controller maximum freedom during normal operation, we decided it would be acceptable for the safety controller to manipulate the wheel angle.

### 3.5.2 Overslip

Overslip is a performance property that becomes violated when the tractor does not operate efficiently due to the wheels slipping. The calculation for slip is straightforward, and given by

\[
\text{slip} = 1 - \frac{\text{accel}_{\text{output}}}{\text{accel}_{\text{input}}}
\]

where \( \text{accel}_{\text{output}} \) is the observed acceleration of the tractor, and \( \text{accel}_{\text{input}} \) is the linear acceleration of the wheels.

The more interesting part of this performance property is the modeling of the slip, the way in which the tractor system’s experienced slip changes over time. We came up with three properties we wanted to contain within our modeling of the slip:

- The tractor should almost always experience some slip, this may be very small or very large, dependent upon the terrain and the force being applied.
- The larger the force applied, the more slip is experienced, on all terrains. The increase in slip is slow for small forces, and faster for larger forces.
- The peak efficiency occurs between 5% and 20% slip, based on tractor parameters.

Based on these properties, we came up with a quadratically increasing slip model, as shown in Figure 3.4. In this model, each surface has an \( x\text{-intercept} \) value which defines the slip on that surface as a function of force. This slip is given by

\[
\text{slip}(x\text{-intercept, accel}_{\text{input}}) = \min \left(1, \left(\frac{\text{accel}_{\text{input}}}{x\text{-intercept}}\right)^2\right)
\]

where \( \text{accel}_{\text{input}} \) is the linear acceleration of the wheels and \( x\text{-intercept} \) is a value specific to the particular surface on which we are driving. Although \( x\text{-intercept} \) is not used in the violation equation, it is used in the model, and therefore must be calculated. Given an \( \text{accel}_{\text{input}} \) (linear wheel velocity), and \( \text{accel}_{\text{output}} \) (observed tractor velocity), we can compute \( x\text{-intercept} \) by setting the actual slip, \( 1 - \frac{\text{accel}_{\text{output}}}{\text{accel}_{\text{input}}} \), equal to the slip predicted by the terrain-based slip equation, \( \left(\frac{\text{accel}_{\text{input}}}{x\text{-intercept}}\right)^2 \). Solving for \( x\text{-intercept} \), we get

\[
x\text{-intercept} = \sqrt[3]{\frac{\text{accel}_{\text{input}}^3}{\text{accel}_{\text{input}} - \text{accel}_{\text{output}}}}
\]
which can then be used to calculate the proper \( x\)-intercept for the least and most slippery surfaces, with the surfaces in between being interpolated linearly.

![Graph showing the quadratically increasing slip model](image)

Figure 3.4: The quadratically increasing slip model was used within the overslip performance property modeling.

In the model-checking search, we compare the current state’s slip value, \( 1 - \frac{\text{accel}_{\text{output}}}{\text{accel}_{\text{input}}} \) with an efficiency threshold, and, if the slip is unacceptably high, the property is considered violated. The safety controller’s command always reduces the force applied by reducing \( \text{accel}_{\text{input}} \), which according to our model will reduce slip.

Notice that this slip model is may not exactly correspond with one intuitive notion of slip, namely that it checks for slip in the acceleration, but not the velocity. Indeed, if the acceleration is zero, but the velocity is nonzero, the tractor, according to this model, is not experiencing any slip. This decision was based on the notion that slip is related to the force that was applied, which corresponds to the acceleration. Even accepting that slip is a function of force, there is another unanticipated property of this slip model, which was not discovered until after we were determining the decision logic switching behavior. This will further be discussed in the overslip behavior determination in Section 3.6.2.

### 3.5.3 Oversteer

Oversteering was the last property for which we applied Simplex. This property is violated when, for example, the wheels are turned at 45 degrees, but the actual tractor is turning at a much lower angle, say 20 degrees. This is usually caused by the conditions on the ground, such as a loose sand surface, or muddy ground. The corrective action in this case is counterintuitive in that we can obtain a lower skid value by reducing the steering angle, whereas increasing the steering angle will likely increase skid. Doing this, however, may be in conflict with higher-level control that only cares about the resultant steering angle. Despite this, for tractor efficiency we do want to prevent excessive skid from being experienced by the tractor. Thus, the corrective action to prevent oversteer is to
straighten the wheels.

In terms of modeling, we used a similar skid model to the one used in overslip shown previously in Figure 3.4. However, since the wheels are now slipping at an angle, the $x$-intercept for the surfaces is likely to be different. We therefore require separate measurements in the model generation file to calculate proper $x$-intercepts for the oversteer property.

### 3.5.4 Collision

Avoiding collisions was also considered as a property for which we could apply Simplex. However, after our initial investigation we decided that it did not easily fit the Simplex framework.

In particular, the biggest issue with collision is discerning obstacles which the tractor should go around, such as large boulders, and objects that the tractor could pass through, such as tall grass. Although the complex controller makes such a distinction using machine learning techniques, we want to provide a verifiable system. In this case, the safety controller would have to be as complicated as the complex controller, there was no benefit from the point of view of verification effort to include it.

However, there are some indirect collision-based properties that we could apply Simplex towards. For example, if the complex controller is not trusted completely we may set a maximum speed which we are allowed to pass through detected obstacles, in order to minimize potential damage in the case of a misclassification. In this case, the simple controller would only need to detect that objects exist in the path of the tractor, not necessarily what type of objects are there.

### 3.6 Behavior Determination

The next step in the industrial system-level Simplex process is to determine when to switch to the safety controller. In order to make this determination, we leverage on the Maude model-checking tool.

Given a generated Maude model, and the properties we are interested in monitoring, we can already run Maude model-checking searches and detect that an unconstrained model, one that always uses the complex controller, results in property violations. Furthermore, we can print out all the reachable states that violate our properties, the static state violations, to a text file and display that information graphically to gain insight into the violation region.

Notice, however, that these states are only the states where a violation is actively occurring, which is a subset of the states we should never enter. Particularly, there may be states that, although themselves do not violate properties, may result in future violations even if we use the safety controller. These dynamic state violations should also be avoided. We could use Maude to calculate these states by, at each state we enter, performing a sub-search which uses only the simple controller, and seeing if the property condition can become violated for all possible external changes and disturbances. Ideally, the user would not need to input any extra information here since our model-checking tool can explore this subset of the model completely. However, in order to speed up the search, we instead relied on the user to provide a worst-case change in the system state. This is not as difficult as it first appears, as it is clear a tractor is more likely to rollover if the slope of the terrain increases in the future, or that we are more likely to experience overslip if the terrain
becomes more slippery in the future. After the user provides these worst-case model transitions, we use Maude to search for all states where, if we were to enter them, even the safety controller could not guarantee that the property would not be violated. This is exactly the information needed to drive the decision module. We will now describe this in slightly more detail for the three properties we checked: rollover, overslip, and oversteer.

### 3.6.1 Rollover

In order to determine the rollover switching logic, we needed to input to Maude two pieces of information: the behavior of the safety controller, and the worst-case change in the nonactuated system state. Our rollover model has three dimensions, the velocity of the tractor (pessimistically assuming a worst-case friction scenario for breaking), the slope of the terrain, and the steering angle of the wheels. Any dimensions not directly modified by the safety controller would be part of the worst-case system state change.

There were two choices for the safety controller, one which slowed the tractor down and straightened the wheels, and the other one which only slowed down the tractor. Initially, we considered the safety controller that just slowed the tractor down. However, in this case the worst-case state change would increase the steering angle of the wheels, while at the same time increasing the slope of the terrain. This turned out to lead to an extremely pessimistic decision module which could not allow the tractor to reach full speed on flat ground with straight wheels, since it would not be able to slow down in time if the complex controller turned the wheels sharply to one side and the terrain slope increased. For this reason, we instead used the safety controller that both straightened the wheels and slowed down the tractor. The worst-case environmental change, then, was just an increase in the terrain’s slope. The associated Maude code, which recursively defines the `can-lead-to-violation` function, is shown in Code Block 1. The first two rewrite equations define the non-recursive case where the state does, and does not violate the rollover condition, respectively. The last rewrite equation is the recursive call which makes use of the `reduce-vel` and `reduce-steerAngle` rewrite equations for the safety controller, as well as the `increase-slope` rewrite equation for the worst-case state change.

**Code Block 1** This Maude code is used to determine which states may lead to future violations even if the safety controller is used.

```maude
eq can-lead-to-violation(vel(0) steerAngle(0) slope(Slope)) = false .

ceq can-lead-to-violation(vel(Vel) steerAngle(Steer) slope(Slope)) = true
   if roll-force(advance-vel(Vel) advance-steerAngle(Steer)
      advance-slope(Slope)) >= 9810.0 .

eq can-lead-to-violation(vel(Vel) steerAngle(Steer) slope(Slope)) =
   can-lead-to-violation(reduce-vel(Vel) reduce-steerAngle(Steer)
      increase-slope(Slope)) [owise] .
```

Using this information, we were able to use Maude to search the entire model for all states where `can-lead-to-violation` evaluates to `true`. Again, these are the states where, if a worst-case state change occurs, even using the safety controller can not guarantee that rollover will not occur. These dynamic state violations can then be plotted graphically. For rollover, we plotted these states on
three 2D graphs\(^2\). Next, we determined linear bounds on the region of states that should be avoided. Although we did this operation manually, it could have been automated by using an \(n\)-dimensional convex hull algorithm [13]. These linear bounds can then be used to drive the decision module behavior. The dynamic state violation graph, as well as the linear bounds we used, are shown in Figure 3.5.

![Rollover Dynamically Unsafe States](image)

Figure 3.5: Three sets of linear bounds on the dynamic state violations (shown as points) in the rollover model drive the decision module’s behavior.

### 3.6.2 Overslip

The overslip model also contains three dimensions: acceleration, velocity, and the terrain’s \(x\)-intercept (as defined in the overslip property discussion, Section 3.5.2). This model differs from the rollover model in that the dimensions have dependencies, most notably that velocity depends on acceleration. This means that, when we define the safety controller and worst-case state change, we do not have to provide an explicit modification for the velocity since it’s determined by the acceleration.

The safety controller here is clear, slow down the tractor by modifying the acceleration dimension. The worst-case state change, then, is a decrease in the \(x\)-intercept dimension, which corresponds to the surface becoming more slippery. Our model generator defines the `reduce-accel` rewrite equation automatically, as shown in Code Block 2. This is used to compute the `can-lead-to-violation` function in the overslip model.

**Code Block 2** This Maude code describes a decrease in acceleration, which is used to model the overslip safety controller.

```
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ceq reduce-accel(cur-accel) = cur-accel + (-1500) if cur-accel &gt; 1500 .</code></td>
<td>If the current acceleration is greater than 1500, decrease it by 1500</td>
</tr>
<tr>
<td><code>ceq reduce-accel(cur-accel) = cur-accel + (1500) if cur-accel &lt; -1500 .</code></td>
<td>If the current acceleration is less than -1500, increase it by 1500</td>
</tr>
<tr>
<td><code>eq reduce-accel(cur-accel) = 0 [owise] .</code></td>
<td>Otherwise, set the acceleration to 0</td>
</tr>
</tbody>
</table>
```

As with the rollover model, we use Maude to search for all system states where `can-lead-to-violation` evaluates to \textit{true}, which correspond to the dynamic state violations in the overslip model. Since our property equation has almost no reliance on velocity, we simply ignored that dimension when graphing the dynamic state violations (or more formally, we projected the 3D dynamic state violations onto the velocity/\(x\)-intercept plane, which may give us a slightly

\(^2\)A 3d state graph could alternatively have been used, as we will show with the oversteer condition.
more pessimistic decision module and is therefore allowed). One might even consider eliminating velocity as a dimension within this model, since initially it appears to have no bearing on the property equation. This intuition clarifies the condition that an automated system could use to prune unnecessary dimensions from a larger model: a dimension can be pruned only if it is not mentioned in the property equation, and does not directly affect any dimensions that are within the model. However, this pruning condition is not true in this case, since the changes allowed in the acceleration are based on whether the tractor is moving forwards or backwards (which is determined by the velocity). In hindsight, having a two-state dimension \{forward, backward\} for this discrete state change would have been more efficient than checking all the various velocities.

We chose linear bounds on the dynamic state violations that should be avoided in order to guarantee the property. This is shown in Figure 3.6.

![Overslip Dynamically Unsafe States](image)

Figure 3.6: Linear bounds on the dynamic state violations (shown as points) in the overslip model are used to prevent overslip when the acceleration is positive.

One initially unexpected observation of the dynamic state violations is that, since the initial overslip property equation (as described in Section 3.5.2) uses the square of the acceleration, overslip could occur both when the tractor was speeding up as well as slowing down. This is not because of incorrect modeling, as, for example, anti-lock breaks in cars are actually used to prevent such slipping when acceleration is negative. This was, however, against the informal notion we set out with overslip, so we modified the property condition for overslip to consider only positive accelerations (when the tractor was going in reverse, we proved via model-checking that overslip could never occur even at full negative acceleration, although this is a model-dependent check). This is the reason the linear bounds in Figure 3.6 only bound the dynamic state violation with positive accelerations.
3.6.3 Oversteer

The final property needing decision module behavior is the oversteer condition. The system state in the oversteer model also contains three dimensions: the velocity of the tractor (assuming worst-case breaking), the steering angle of the wheels, and the x-intercept of the terrain. Notice that the x-intercept used in this model is different from the x-intercept in the overslip condition, since in oversteer the wheels are skidding almost perpendicular to the intended direction of the tractor, and are therefore going to experience friction differently.

For the safety controller, we reduce the steering angle and decrease the velocity. This means that the worst-case external state change corresponds to a decrease in the x-intercept (the surface becoming more slippery). The Maude code for this worst-case change, which is produced by our model generator for the specific tractor model we used, is shown in Code Block 3. The decreaseXInt rewrite equation is used to define the can-lead-to-violation function.

**Code Block 3** This generated Maude code recursively defines the worst-case external state change for a particular tractor model for the oversteer condition. When the x-intercept decreases, the modeled tractor experiences increased skidding.

```
op decreaseXInt(_) : Int -> Int .

ceq decreaseXInt(cur-x-intercept) = cur-x-intercept - 1760 
  if cur-x-intercept > 4274 .

eq decreaseXInt(cur-x-intercept) = 4274 [owise] .
```

Next, we use the Maude model checker to find all states within the oversteer model where can-lead-to-violation evaluates to true. These dynamic state violations are plotted in three dimensions in Figure 3.7. Two planes are used to linearly bound the dynamic state violations in three dimensions.

![Oversteer Dynamically Unsafe States](image)

Figure 3.7: Two planes bound the dynamic state violations (shown as points) within the oversteer state space. On the right, a rotated view clearly shows all dynamic state violations residing within the state-space defined by the two planes.
3.7 Behavior Description and VMaude

Now that we have calculated which states should be avoided, we can exactly describe the behavior of the decision module. At each control iteration, the decision module receives commands from the complex controller. Using these, in addition to a worst-case evaluation of nonactuated dimensions, it predicts the future state of the system. If this future state falls outside of the linear bounds we calculated above, if the future state is not a dynamic state violation, the complex controller is used. Otherwise, the safety controller is used.

We can exactly describe the decision module behavior in various ways. One option is to directly code the behavior in Maude, and then perform a model check on the aggregate Simplex controller in the tractor state space to verify that it does not violate the property equations. This is useful for showing the correctness of the decision module, however it does not easily translate into production code. For this reason, we created a Maude module describing the syntax and semantics of VMaude, which is VHDL-structured Maude code. Although VMaude code remains executable within Maude, it has a direct correspondence with a VHDL module. This means that the VMaude version of the decision module can be model-checked, in addition to being easily and automatically translatable to executable VHDL hardware code.

We now present the two behavior description methods and associated model-checking searches. First, we show how to describe the decision module in pure Maude in Section 3.7.1. Next, in Section 3.7.2, we describe VMaude and the VMaude decision modules, as well as the process of formally proving equivalence with the corresponding original Maude decision modules.

3.7.1 Maude Decision Module Behavior Description

The Maude-based decision module consists of two parts. First, a prediction of the future state space is made based on the complex-controller command (if no command is received in a timely manner, a default can be used). This future state is then checked to see if it is in the dynamic state violation space defined by the linear bounds derived in Section 3.6. We go through this process for the three properties we modeled: rollover, overslip, and oversteer.

The Maude code for the Simplex decision module for the rollover condition is shown in Code Block 4. There are two rewrite equations for this code. The first one checks if the predicted worst-case next state, which consists of NewVel, NewSteer and increase-slope(Slope), is within the linear bounds for the dynamic state violations. If it is, the safety controller is used (by setting the next state to reduce-vel(Vel) reduce-steerAngle(Steer) slope(NewSlope)). The second equation uses the complex controller (sets the next state to vel(NewVel) steerAngle(NewSteer) slope(NewSlope)), if the first equation’s condition evaluates to false, as indicated by the [owise] (otherwise) condition.

The linear bounds checked by this module correspond to the lines shown in Figure 3.5. When the steering angle is positive, three sets of bounds (one for each 2D state space in the figure) are given in Equations 3.5, 3.6, and 3.7.
This Maude code describes the decision module within the rollover model.

cqe simplex-evaluate(Vel Steer Slope
  vel(NewVel) steerAngle(NewSteer) slope(NewSlope))
  = reduce-vel(Vel) reduce-steerAngle(Steer) slope(NewSlope)

if ((NewSteer > 0 and
    ((increase-slope(Slope) == slope(600) and NewSteer > 2400 - NewVel quo 5)
    or (increase-slope(Slope) == slope(800) and
        ((NewVel < 5000 and NewSteer > 3557 - (NewVel * 4) quo 7) or
         (NewVel >= 5000 and NewSteer > 700)))
    or (increase-slope(Slope) == slope(1000) and NewVel > 2900 and
        ((NewVel < 4900 and NewSteer > 1770 - (NewVel * 3) quo 10) or
         (NewVel >= 4900 and NewSteer > 300))))
  or (NewSteer < 0 and
    ((increase-slope(- Slope) == slope(600) and
      (- NewSteer) > 2400 - NewVel quo 5)
     or (increase-slope(- Slope) == slope(800) and
        ((NewVel < 5000 and (- NewSteer) > 3557 - (NewVel * 4) quo 7) or
         (NewVel >= 5000 and (- NewSteer) > 700)))
     or (increase-slope(- Slope) == slope(1000) and NewVel > 2900 and
        ((NewVel < 4900 and (- NewSteer) > 1770 - (NewVel * 3) quo 10) or
         (NewVel >= 4900 and (- NewSteer) > 300))))).

eq simplex-evaluate(Vel Steer Slope
  vel(NewVel) steerAngle(NewSteer) slope(NewSlope))
  = vel(NewVel) steerAngle(NewSteer) slope(NewSlope) [wise].

\[
\text{increase-slope}(\text{Slope}) = 600 \text{ and } \text{NewSteer} > 2400 - \frac{\text{NewVel}}{5} \quad (3.5)
\]

\[
\text{increase-slope}(\text{Slope}) = 800 \text{ and } \text{NewVel} < 5000 \text{ and }
\text{NewSteer} > 3557 - \frac{4 \times \text{NewVel}}{7} \text{ or } \text{NewVel} \geq 5000 \text{ and } \text{NewSteer} > 700 \quad (3.6)
\]

\[
\text{increase-slope}(\text{Slope}) = 1000 \text{ and } \text{NewVel} > 2900 \text{ and } \text{NewVel} < 4900
\text{ and } \text{NewSteer} > 1770 - \frac{3 \times \text{NewVel}}{10} \text{ or } \text{NewVel} \geq 4900 \text{ and } \text{NewSteer} > 300 \quad (3.7)
\]

Notice that these are the same equations as in the first half of the condition of the first rewrite equation in the Maude code in Code Block 4. The second half of the first rewrite equation’s condition defines the symmetric cases where the steering angle is negative.

The decision module for the overslip condition is a bit simpler, since the dynamic state violations are defined on one two dimensional state space with no symmetric cases. The condition used to check if can use the complex controller is shown in Equation 3.8.

\[
\text{curXInt} < 1100 \text{ or } \text{nextAccel} < 600 \text{ or } \text{nextAccel} < \left(\frac{5 \times \text{curXInt}}{7} - 614\right) \quad (3.8)
\]

Finally, for the oversteer property, we also create the Maude code describing the behavior of the decision module. The two equations for the two planes that bound the dynamic state violations in
the oversteer model (shown graphically in Figure 3.7) are shown in Equations 3.9 and 3.10. Notice that these plane equations contain floating-point numbers, which are not easily represented in a hardware description language like VHDL. Thus, the associated hardware code uses approximations for these numbers, and it is necessary to prove that these approximations do not alter the behavior of the decision module. Performing this proof is described in the VMaude discussion in Section 3.7.2.

\[
\text{nextSteerAngle} \geq 400 
\]

\[
\text{nextSteerAngle} \geq -0.511 \times \text{nextVel} + 0.153 \times \text{reduceXInt(curXInt)} + 2030 
\]

For any of the models, after we have codified the decision module in Maude, we can run state-space searches to see if property violations can occur. This is used to verify the correctness of the Maude decision module. For oversteer, for example, we can search for a property violation on the aggregate system. The associated rewrite rules and search statement are shown in Code Block 5.

The search statement is searching for static state violations. Informally the search statement is saying that, in the TRACTOR model, starting at the initial state and use zero or more rewrites (\(\Rightarrow^*\)), find all states where the skid at that state is greater than 0.3 (30%). After a few minutes, this search finishes checking the entire model without producing any such states, indicating that the decision module indeed prevents the system from violating the oversteer condition. Similar searches were performed for rollover and overslip.

**Code Block 5** This Maude code searches for static state violations within the oversteer model.

```maude
--- compute angular velocity in non-slip situation
eq angVel(curVel, curSteer) = float(curVel) \times \sin(float(curSteer) / 1000.0) / float(WHEEL-BASE) .

--- compute skid
eq skid(curVel, curSteer, curXint) = ((angVel(curVel, curSteer) \times float(curVel)) / float(curXint)) \times ((angVel(curVel, curSteer) \times float(curVel)) / float(curXint)) .

--- search for static state violations
search in TRACTOR : initial
\(\Rightarrow^*\)
vel(Vel:Int) x-intercept(XInt:Int) steerAngle(SteerAngle:Int)
such that
skid(Vel:Int, SteerAngle:Int, XInt:Int ) > 0.3 .
```

### 3.7.2 VMaude Decision Module Behavior Description

The eventual output of the System-Level Simplex Industrial case study should be VHDL code for the verified decision module. In order to minimize coding errors between the verifiably correct Maude version of the decision module and the final implementation, we strive to generate this final code automatically. For this reason we provide an intermediate representation called VMaude, or VHDL-structured Maude, which, although written and executable within Maude, has a direct correspondence to VHDL code.

First, we describe VMaude and show some example formal semantics for the language. Second,
we show an example decision module written in VMaude, which remains executable within Maude and serves to demonstrate some of the restrictions while using VMaude. Last, we formally prove an equivalence between the VMaude decision module and one of the previously-developed pure Maude decision modules.

VMaude contains a number of simple constructs which, when combined, can be used to perform the linear bounds operations necessary for the decision module. It is not a complete definition of all the semantics of VHDL within Maude, but rather, it defines enough VHDL semantics to do something useful, to perform the decision module’s linear bounds operations.

That said, there are two main types of statements in VMaude. The first kind is used to define signals(variables), their direction (input / output / internal), and their bit lengths. The second kind is used to define the finite state machine that performs logical operations. The state machine has some implicit behavior. It is in state 0 initially and when idle, and transitions to state 1 when the 'start signal is asserted. When the state machine transitions back into state 0 (presumably after performing some computation), the 'done signal is asserted for one clock cycle. Additionally, the user must define a 'fsmState internal signal with an appropriate bit length, which is used to store the value of the current state of the finite state machine. The actual finite state machine behaves according to the user-provided VMaude statements. The following list includes all currently allowed VMaude state machine statements:

- **assignInt**: assigns an integer to a signal and deterministically transitions to another state
- **assignSignal**: assigns a signal to another signal and deterministically transitions to another state
- **assignAddSignalInt**: assigns the sum of a signal and an integer to another signal and deterministically transitions to another state
- **assignAddSignalSignal**: assigns the sum of a signal and another signal to yet another signal and deterministically transitions to another state
- **assignMultSignalInt**: assigns the product of a signal and an integer to another signal and transitions to another state
- **assignShiftRightSignalInt**: assigns the value of a signal shifted logically to the right by an integer to another signal and deterministically transitions to another state
- **conditionalGreaterSignalInt**: branches to one of two provided next states, based on whether a provided signal is greater than a provided integer
- **conditionalEqualsSignalInt**: branches to one of two provided next states, based on whether a provided signal is equal to a provided integer
- **conditionalGreaterSignalSignal**: branches to one of two provided next states, based on whether a provided signal is greater than another provided signal
- **conditionalLesserSignalInt**: branches to one of two provided next states, based on whether a provided signal is less than a provided integer
By combining these statements, we are able to create the decision modules for all the properties within our case study. In order to be executable within Maude, however, these statements need corresponding Maude semantics. Consider the conditionalGreaterSignalInt statement. The formal semantics for this statement in Maude are shown in Code Block 6.

**Code Block 6** This Maude code formally defines the semantics of the VMaude conditionalGreaterSignalInt statement.

```maude
--- Conditional Greater Signal Int
c eq iterate( Lines conditionalGreaterSignalInt(StateNum VarName IntValue
   TrueNextStateNum FalseNextStateNum) signal-value('fsmState Length StateNum) ) =
   iterate( incrementCycleCount(
      setValue( Lines conditionalGreaterSignalInt(StateNum VarName IntValue
       TrueNextStateNum FalseNextStateNum)
      signal-value('fsmState Length StateNum)
      , 'fsmState , TrueNextStateNum)
   )
   if getInternalValue(VarName, Lines) > IntValue .

ec iterate( Lines conditionalGreaterSignalInt(StateNum VarName IntValue
   TrueNextStateNum FalseNextStateNum) signal-value('fsmState Length StateNum) ) =
   iterate( incrementCycleCount(
      setValue( Lines conditionalGreaterSignalInt(StateNum VarName IntValue
       TrueNextStateNum FalseNextStateNum)
      signal-value('fsmState Length StateNum)
      , 'fsmState
      , FalseNextStateNum)
   )
   [otherwise].
```

Particularly, notice that if the first rewrite equation executes, if the condition getInternalValue(VarName, Lines) > IntValue evaluates to true, then we do a setValue on the 'fsmState signal to TrueNextStateNum. If the first rewrite equations does not match, the second one will (since the second one’s condition is [otherwise]). In the second rewrite equation, we do a setValue on the 'fsmState signal to FalseNextStateNum. These two equations then together provide the executable Maude semantics for the conditionalGreaterSignalInt VMaude statement. Similar semantics were written for the nine other VMaude state machine statements, and are available in our code reference in Appendix A.4.

The next step is to write the decision module in VMaude. This is a manual process, however, any errors made at this step can be detected by the Maude engine, as we will later demonstrate. The manually-coded VMaude decision module for the oversteer property is shown in Code Block 7. For reference, we have also included the rollover VMaude code in Appendix A.5, and the corresponding VHDL code in Appendix A.6.

Notice that VMaude is less expressive than pure Maude. For example, VMaude cannot deal with floating-point numbers directly (which makes sense since neither can VHDL) and thus fractions must be used to provide appropriate approximations. Additionally, arbitrary division is not allowed, but instead only bit shifts can be performed which effectively limits us to dividing by powers of two. In the sample code in Code Block 7, starting at state 9, we can not use the floating point value -0.511, so instead we end up multiplying by -131, and then shifting to the right by 8 (which is effectively a division by 256). Hence, \( \frac{-131}{256} \) is used as an approximation for -0.511. One concern with
This VMaude program evaluates the decision module for the oversteer property.

--- input signals
(input 'iNextVel 16)
(input 'iCurXInt 16)
(input 'iNextSteerAngle 16)

--- output signals
(output 'allow 2) --- 1 = allow, -1 = disallow

--- internal signals
(signal 'fsmState 8)
(signal 'steerAngle 32)
(signal 'nextXInt 16)
(signal 'rhs 32) --- right hand side
(signal 'rhs2 32) --- right hand side temporary value

--- first we make steerAngle positive
(state 1 assignSignal 'steerAngle 'iNextSteerAngle 2)
(state 2 conditionalGreaterSignalInt 'steerAngle 0 4 3)
(state 3 assignMultSignalInt 'steerAngle 'steerAngle -1 4)

--- check 1: iNextSteerAngle < 400
(state 4 conditionalLesserSignalInt 'steerAngle 400 20 5)

--- next compute nextXInt
(state 5 assignSignal 'nextXInt 'iCurXInt 6)
(state 6 assignAddSignalInt 'nextXInt 'nextXInt -1760 7) --- nextXInt -= 1760
(state 7 conditionalGreaterSignalInt 'nextXInt 4274 9 8) --- lower bound by 4274
(state 8 assignInt 'nextXInt 4274 9)

--- now allow if curSteer < -0.511 * Vel + 0.153 * Xint + 2040
--- -0.511 = -131 / 256 ; 0.153 = 313/2048
(state 9 assignSignal 'rhs 'iNextVel 10)
(state 10 assignMultSignalInt 'rhs 'rhs -131 11)
(state 11 assignShiftRightSignalInt 'rhs 'rhs 8 12)

(state 12 assignSignal 'rhs2 'nextXInt 13)
(state 13 assignMultSignalInt 'rhs2 'rhs2 313 14)
(state 14 assignShiftRightSignalInt 'rhs2 'rhs2 11 15)

(state 15 assignAddSignalSignal 'rhs 'rhs 'rhs2 16)
(state 16 assignAddSignalInt 'rhs 'rhs 2040 17)

(state 17 conditionalGreaterSignalSignal 'rhs 'steerAngle 20 21)

--- set 'allow output signal
(state 20 assignInt 'allow 1 0) --- 'allow command; done
(state 21 assignInt 'allow -1 0) --- 'disallow command; done
this is that $\frac{-131}{256}$ is not exactly equal to -0.511, so that the resultant decision module may not behave
the same as the floating-point based Maude decision module. We now address this important issue.

Manually translating from the pure Maude decision module to a VMaude version introduces
two sources of errors. First, as described above, approximations may have to be used to overcome
VMaude restrictions (which are there because of VHDL restrictions). Second, manually coding the
module my introduce other bugs because it is not an automatic process. Ideally we would want to
prove that the decision made by the VMaude decision module is always the same as the pure Maude
decision module.

Since VMaude has formal Maude semantics, we can leverage on the Maude engine to prove this
for us. For every state in our tractor model, we can compare the output of the pure Maude decision
module and the VMaude decision module. If any discrepancies are found, they are reported. In
practice, we came across both approximation errors (as in the oversteer VMaude code) and manual
coding errors such as transitioning to an unintended state. Since the Maude engine gives us the
parameters where the decisions diverge, debugging the VMaude code is greatly simplified. When
the Maude search no longer reports violations, the two modules are behaviorally identical for every
state in the model.

The way this was done was, at each state, Maude checks the outputs of the two decision mod-
ules. If the outputs are different, we transition to a state with the DifferentBehavior operator,
which contains the values of the dimensions for the current state and the proposed next state.
Since there are no semantics defined for the DifferentBehavior operator, it will stop Maude
from further rewriting the term. The search, then, simply looks for any states which contain the
DifferentBehavior operator. The search statement for the oversteer module is shown in Code
Block 8.

**Code Block 8** This search checks for divergent behavior between the pure Maude decision module
and the VMaude decision module.

--- search for instances of different behavior
search in TRACTOR : initial
    ==> DifferentBehavior(nextVel:Int nextSteerAngle:Int nextXInt:Int

3.8 Code Generation

The final step of the Industrial System-Level Simplex case study to generate VHDL code which
can directly run on FPGA hardware. Since the decision module is already written in VMaude, this
process is straightforward. In order to eliminate errors from manual translation, we have written a
Java program which performs this translation automatically. Although this program is not formally
verified, the translation is straightforward, and does not contain complex logic. We have included
the source code for reference in Appendix A.7.

For example, consider the VMaude conditionalGreaterSignalInt statement. This statement
takes in two potential next states, and branches to one of them depending on the value of a provided
signal and a provided integer. An input VMaude statement is shown in Code Block 9. The portion
of the Java code of the translator associated with this particular translation is in Code Block 10, which produces the final VHDL code presented in Code Block 11.

**Code Block 9** VMaude statements such as this one are input into the Java-based VHDL generator, and processed by the code in Code Block 10

```
(state 2 conditionalGreaterSignalInt 'worstCaseSlope 1000 3 4)
```

**Code Block 10** Java code translates the VMaude `conditionalGreaterSignalInt` statement to VHDL code, resulting in the VHDL code shown in Code Block 11.

```
else if (tokens[2].equals("conditionalGreaterSignalInt"))
{
    String sig = tokens[3].substring(1);
    fout.write("if " + sig + " > " + tokens[4] + " then\n");
    fsmTransition(3,tokens[5],fout);
    fout.write("\t	else\n");
    fsmTransition(3,tokens[6],fout);
    fout.write("\t	end if;\n");
}
```

**Code Block 11** This VHDL code is produced by the Java-based generator from the VMaude code in Code Block 9.

```
when 2 =>
    if worstCaseSlope > 1000 then
        fsmState <= 3;
    else
        fsmState <= 4;
    end if;
```

The final output of the code generator is a complete VHDL module, which can be immediately synthesized and run on hardware. We took a generated VHDL module for the decision module associated with the rollover property and simulated it directly using Xilinx’s ISE Simulator. This tool has no knowledge of Maude or VMaude, and directly simulates the VHDL Code. The simulation trace is shown in Figure 3.8. In the testbench for this simulation, at 100 ns, four things occur. The `start` signal is asserted for one clock cycle, the value of 200 milliradians is set as to `slope`, the `nextSteering` signal is set to 1400 milliradians, and the future state velocity, `nextVelocity`, is set to 6000 millimeters per second. Over the next several clock cycles, the value of `fsmState` changes as the decision module goes through the finite state machine. At time 170 ns, the state machine finishes and transitions back to state 0. The `done` signal is asserted, and the `allow` output signal is also asserted, indicating that the complex controller should be allowed to actuate the system.

### 3.9 Improving Model-Checking Run Time

The potential model-checking time of a system varies widely, from fractions of a second to hundreds of years, if done incorrectly. This section provides insight into managing model-checking run time. We first disclose methods used to significantly reduce the model-checking time during development, and later describe future ideas for performing this important operation.
We used four techniques to reduce the size of the state space, and therefore decrease model-checking time. First, we took advantage of symmetry within our models. For example, in the rollover model, whether the wheels are turning left or right only affects the signs of variables within our rollover equation. Therefore, we can immediately eliminate half of the state space by only model-checking nonnegative steering angles. For the final check, however, we must restore the model to the complete state space. Second, we statically checked whether properties could be violated in certain situations, and pruned parts of the model if it was always safe. For example, in the overslip model, we calculated that even at maximum velocity, the tractor could not experience overslip while in reverse. Therefore, we removed any negative velocities from our model. Third, we eliminated entire dimensions through careful manipulation. Both in the rollover model, and in the oversteer model, we do not model acceleration, but instead only consider velocity and assume worst-case breaking. This significantly reduced model-checking time since there was one less dimension. The fourth technique we used, if the model-checking time was still unacceptably high, was to increase the discretization constants. This essentially creates a more pessimistic decision module, however, large enough discretization constants will always result in a tractable state space.

In the future, we can also consider to add other means to reduce model-checking time. One radical approach we started investigating was, instead of checking the entire model offline, running a hardware model-checker online, and checking some amount of time into the future of the current state. Then, if no violations are detected several seconds into the future (and we proved this was all we needed to check for this particular model), the complex controller can be used. In this way, the entire state space, even if unmanageably large, never needs to be checked. The drawbacks of this is that a generic, real-time hardware model checker does not exist, so we instead strived to generate
specific hardware code for the particular property we were using. The other promising future approach for reducing model-checking time is the use of model abstractions. Model abstractions provide a way to collapse several states of a system into a single state, while preserving the model-checking result. This is shown abstractly in Figure 3.9. Here, the grey states are states which are externally guaranteed to not violate the property condition, whereas the white ones may represent states where violations may occur. Since no violation states are abstracted onto a nonviolation state, we can say the abstraction is \textit{invariant preserving}, and will therefore preserve the result of model checking (this is the same reason we can take an infinite continuous dimension and model it as a finite number of discrete states). In the future, we plan to investigate ways in which these abstractions can be automatically derived, or externally proven and input by the user.

Figure 3.9: Model abstractions can collapse several states into a single one, improving model-checking run time. Here, the grey states on the left are collapsed into a single state on the right.
Chapter 4

Conclusions

The future trend of computing is an increased interaction with physical environments. These cyber-
physical systems, however, must be designed carefully in order to prevent monetary losses and, more
importantly, avoid bodily harm. In this thesis, we have significantly progressed the state-of-the-art
by addressing key issues with the Simplex Architecture.

We first described the System-Level Simplex Architecture, a novel framework which protects
the system against a superset of faults when compared with the original Application-Level Simplex
Architecture. We reduced the potential for failure of the composite Simplex system by moving the
safety controller and decision module to dedicated hardware. The novel idea here is performing
hardware/software co-design, but not for the classical reasons of power savings or performance
optimization, but rather to provide system safety.

We also went through an in-depth case study, which used model checking to produce a verifi-
ably correct Simplex safety core. Unlike previous control theory techniques, our model checking
approach is easily applicable to discrete and hybrid (mixed discrete and continuous) systems. This
is important, since we expect a large portion of cyberphysical systems to be best modeled as hybrid
systems.

In the future, we plan to build upon both of these key results.

Hardware/software co-design, we believe, is applicable to more than just the classical uses for
power and performance. In addition to safety, we believe a hardware processor is an ideal place for
certain security checks. In one proposed attack, a timing covert channel was created by hardware,
which would extract keyboard input and diverge collected information by minutely tweaking the
timing of network packets [22]. A hardware block could receive this information prior to forwarding
it to the network card and examine the precise timing of the data stream for potential covert
channels.

In terms of model-checking for Simplex correctness, a number of additions are planned. First, to
increase the applicability of our approach, we will allow well-known formats such as Simulink models
to input system dynamics and safety controller behavior. We also plan to examine the composability
problem where multiple safety properties are monitored by Simplex concurrently. This is particularly
challenging if the safety controller behaviors are not always identical. Another direction here is to
decrease the model-checking time by automatically deriving invariant preserving abstractions, which
will effectively collapse a large number of states in the model to a smaller number which can be
checked more rapidly by the Maude engine.
Appendix A

Code Reference

This appendix lists the source code several important programs we developed while investigating the System-Level Simplex Architecture.

A.1 Inverted Pendulum VHDL Code

The following VHDL file implements the main logic for the System-Level Simplex decision module and safety controller.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity io_logic is
Port (data_present : in std_logic;
      clk : in STD_LOGIC;
      input_bits : in STD_LOGIC_VECTOR (7 downto 0);
      output_bits : out STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
      send_now : out STD_LOGIC := '0';
      led_signal : out STD_LOGIC := '0';
      led_signal2 : out STD_LOGIC := '0';
      mem_volts : in std_logic_vector (31 downto 0);
      mem_updated : in std_logic_vector (31 downto 0);
      mem_track : out std_logic_vector (31 downto 0) := (others => '0');
      mem_angle : out std_logic_vector (31 downto 0) := (others => '0'))
);

architecture Behavioral of io_logic is

component hr_controller port (start : in std_logic;
clock : in std_logic;
angle : in std_logic_vector(31 downto 0);
track : in std_logic_vector(31 downto 0);
cx_vols : out std_logic_vector(31 downto 0);
output_avail : out std_logic;

--------------- FPU signals
start_i : out std_logic;
ops_i : out std_logic_vector(31 downto 0);
opsj : out std_logic_vector(31 downto 0);
fpui : out std_logic_vector(2 downto 0);
rmode_i : out std_logic_vector(1 downto 0);
output_o : in std_logic_vector(31 downto 0);
ready_o : in std_logic
```
--- FPU signals
signal ri_start : std_logic := '0';
signal ri_volt_in : std_logic_vector(31 downto 0) := (others => '0');
signal ri_volt_out : std_logic_vector(31 downto 0) := (others => '0');
signal ri_output_avail : std_logic := '0';
signal ri_range_max : std_logic_vector(31 downto 0) := (others => '0');
signal ri_angle : std_logic_vector(31 downto 0) := (others => '0');
signal ri_track : std_logic_vector(31 downto 0) := (others => '0');
signal ri_in_range : std_logic := '0';
--- ri_fpu_outputs
signal ri_opa_i : std_logic_vector(31 downto 0);
signal ri_opb_i : std_logic_vector(31 downto 0);
signal ri_opc_i : std_logic_vector(2 downto 0);
signal ri_rmode : std_logic_vector(1 downto 0);
--- other signals
signal lastmem_updated : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal send_clock_count : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal valid : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal received_track : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal received_angle : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal temp_received_track : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal temp_received_angle : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal high_rel_track : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal high_rel_angle : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal send_vols : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal final_vols : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal range_limited : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
type reset_states is (got0, got1, got2, got3);
signal reset_state : reset_states := got0;
signal restart_receive : std_logic := '0';
signal second_count : integer := 0;
signal controller_shutdown_time : integer := 0;
--- signals for the high reliability controller
signal hr_start : STD_LOGIC := '0';
signal hr_vols : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal hr_output_avail : STD_LOGIC := '0';
type receive_states is (receive_track1, receive_track2, receive_track3,
receive_track4, receive_angle1, receive_angle2, receive_angle3, receive_angle4,
restart fsm_start, wait_for_reset);
signal receive_state : receive_states := wait_for_reset;

--- hr fpu outputs
SIGNAL hr_start_i : std_logic := '0';
SIGNAL hr_opa_i : std_logic_vector(31 downto 0);
SIGNAL hr_opb_i : std_logic_vector(31 downto 0);
SIGNAL hr_fpu_op_i : std_logic_vector(2 downto 0);
SIGNAL hr_rmode_i : std_logic_vector(1 downto 0);

--------- fpu below
SIGNAL fpu_start_i : std_logic := '0';
SIGNAL fpu_opa_i : std_logic_vector(31 downto 0);
SIGNAL fpu_opb_i : std_logic_vector(31 downto 0);
SIGNAL fpu_fpu_op_i : std_logic_vector(2 downto 0);
SIGNAL fpu_rmode_i : std_logic_vector(1 downto 0);
SIGNAL fpu_start_o : std_logic_vector(31 downto 0);
SIGNAL fpu_opa_o : std_logic;
SIGNAL fpu_opb_o : std_logic;
SIGNAL fpu_fpu_op_o : std_logic;
SIGNAL fpu_rmode_o : std_logic;
SIGNAL fpu_start_e : std_logic;
SIGNAL hr_start_e : std_logic;
SIGNAL hr_fpu : std_logic;
SIGNAL fpu_ready_o : std_logic;
SIGNAL fpu_in_e : std_logic;
SIGNAL fpu_overflow_o : std_logic;
SIGNAL fpu_div_zero_o : std_logic;
SIGNAL fpu_inf_o : std_logic;
SIGNAL fpu_zero_o : std_logic;
SIGNAL fpu_qnan_o : std_logic;
SIGNAL fpu_snan_o : std_logic;

--- semaphorish stuff
SIGNAL start_final_range_check_state : integer := 10;
SIGNAL start_reliable_controller_state : integer := 10;
SIGNAL semaphore_checking_final_range : std_logic := '0';
SIGNAL semaphore_running : std_logic := '0';

--- slow clock
SIGNAL slow_clock : std_logic := '0';

--- constants
constant CONSTANT_5_0 : std_logic_vector(31 downto 0) := X"409eb852";
constant CONSTANT_m5_0 : std_logic_vector(31 downto 0) := X"c09eb852";
constant CONSTANT_10_0 : std_logic_vector(31 downto 0) := X"41200000";
constant CONSTANT_m10_0 : std_logic_vector(31 downto 0) := X"c1200000";

COMPONENT fpu
PORT(
  clk_i : IN std_logic;
  opa_i : IN std_logic_vector(31 downto 0);
  opb_i : IN std_logic_vector(31 downto 0);
  fpu_op_i : IN std_logic_vector(2 downto 0);
  rmode_i : IN std_logic_vector(1 downto 0);
  start_i : IN std_logic;
  output_o : OUT std_logic_vector(31 downto 0);
  ready_o : OUT std_logic;
  ine_o : OUT std_logic;
  overflow_o : OUT std_logic;
  underflow_o : OUT std_logic;
  div_zero_o : OUT std_logic;
  qnan_o : OUT std_logic;
  snan_o : OUT std_logic);
END COMPONENT;

begin
my_fpu : fpu PORT MAP(
  clk_i => slow_clock,
  opa_i => fpu_opa_i,
  opb_i => fpu_opb_i,
  fpu_op_i => fpu_fpu_op_i,
  rmode_i => fpu_rmode_i,
  output_o => fpu_output_o,
  start_i => fpu_start_i,
  ready_o => fpu_ready_o,
  ine_o => fpu_ine_o,
  overflow_o => fpu_overflow_o,
  underflow_o => fpu_underflow_o,
  div_zero_o => fpu_div_zero_o,
  inf_o => fpu_inf_o
);
zero_o \Rightarrow \text{fpu.zer_o}
qnan_o \Rightarrow \text{fpu.qnan_o}
snan_o \Rightarrow \text{fpu.snan_o})

\text{p.increment.input : process(clk) begin}
\text{if rising_edge(clk) then}
\text{if slow_clock = '0' then}
\quad \text{slow_clock <= '1';}
\text{else}
\quad \text{slow_clock <= '0';}
\text{end if;}
\text{if controller_shutdown_time > 0 then}
\quad \text{controller_shutdown_time <= controller_shutdown_time - 1;}
\text{end if;}
\text{if received_data = x"00000000" then}
\quad \text{led_signal <= '1';}
\text{else}
\quad \text{led_signal <= '0';}
\text{end if;}
\text{(we need to send at 50 hz and one second is 33000000 ticks)}
\text{(57600 bps)}
\text{if send_clock_count >= 330000 then -- was 660000}
\quad \text{send_clock_count <= (others => '0');}
\text{else}
\text{send_reset_byte 1/4}
\text{if send_clock_count = 1 then}
\quad \text{send_now <= '0';}
\text{else if send_clock_count < 1000 then}
\quad \text{send_now <= send_clock_count + 1;}
\text{else}
\quad \text{send_now <= 0;}
\text{end if;}
\quad \text{output_bits <= x"ff";}
\text{elsif send_clock_count = 10312 then}
\text{elsif send_clock_count = 10313 then}
\text{(send reset byte 2/4)
\quad \text{send_now <= '1';}
\text{elsif send_clock_count = 20624 then}
\text{elsif send_clock_count = 20625 then}
\text{(send reset byte 3/4)
\quad \text{send_now <= '0';}
\text{elsif send_clock_count = 30937 then}
\text{elsif send_clock_count = 30938 then}
\text{(send reset byte 4/4)
\quad \text{send_now <= '1';}
\text{elsif send_clock_count = 41249 then}
\text{(send_now <= '0';)
\text{elsif send_clock_count = 41250 then}
\text{(get the latest value for the volts we're sending)
\quad \text{send_volts <= final_volts_out;}
\text{elsif send_clock_count = 51562 then}
\text{elsif send_clock_count = 51563 then}
\text{(send data byte 1/4)
\quad \text{send_now <= '1';}
\text{elsif send_clock_count = 51564 then}
\text{elsif send_clock_count = 61874 then}
\text{(send_now <= '0';)
\text{elsif send_clock_count = 61875 then}
\text{(send data byte 3/4)
\quad \text{send_now <= '1';}
\text{elsif send_clock_count = 72187 then
\text{(send_now <= '0';)
\text{elsif send_clock_count = 72188 then

43
send_now <= '0';
elsif send_clock_count <= 72188 then
    send_data_byte 4/4
    send_now <= '1';
output_bits <= send_volts(31 downto 24);
elsif send_clock_count >= 82499 then
    send_now <= '0';
end if;
send_clock_count <= send_clock_count + 1;
end if;

-- HIGH RELIABLE CONTROLLER CODE BELOW

case start_reliable_controller_state is
when 0 =>
    semaphore_running_hrc <= '1';
    start_reliable_controller_state <= 1;
when 1 =>
    -- this is false if there was a collision
    start_reliable_controller_state <= 2;
    hr_start <= '0';
    when 2 =>
        hr_start <= '1';
        start_reliable_controller_state <= 3;
    when 3 =>
        hr_start <= '0';
        start_reliable_controller_state <= 4;
        led_signal2 <= '1';
        when others =>
            end case;
if hr_output_avail = '1' then
    led_signal2 <= '0';
    final_volts_out <= hr_volts;
    semaphore_running_hrc <= '0';
    start_reliable_controller_state <= 10;
end if;

-- HIGH RELIABLE CONTROLLER CODE ABOVE

-- FINAL RANGE CHECK BELOW

case start_final_range_check_state is
when 0 =>
    -- store them once here so they don't get changed if we update
    rl_angle <= received_angle;
    rl_track <= received_track;
    when 1 =>
        if semaphore_running_hrc = '0' then
            start_final_range_check_state <= 1;
        end if;
        when 2 =>
            rl_start <= '0';
            when 3 =>
                -- check angle
                rl_range_max <= CONSTANT10;
                rl_range_min <= CONSTANTm10;
                rl_volt_in <= rl_angle;
                rl_start <= '1';
                start_final_range_check_state <= 4;
        when 4 =>
            rl_start <= '0';
            when 5 =>
                -- stay in stage 4 means we're checking angle
                when 8 =>
                    -- 8 means envelope means check final range
                    rl_range_max <= CONSTANT15;
                    rl_range_min <= CONSTANTm15;
                    rl_volt_in <= final_volts_out;
                    rl_start <= '1';
                    start_final_range_check_state <= 9;
        when 9 =>
            -- 9 means we're doing final range limiting
            rl_start <= '0';
            when others =>
                end case;
if \( r_{l\text{output avail}} = '1' \) then
  if \( \text{start.final_range_check.state} = 2 \) then
    -- we just checked track
    if \( r_{l\text{in range}} = '1' \) then
      \text{start.final_range_check.state} <= 3;  -- check angle
    else
      \text{start.final_range_check.state} <= 8;  -- use value from reliable controller
    end if;
  end if;
  \text{controller.shutdown.time} <= 165000000;
end if;
else if \( \text{start.final_range_check.state} = 4 \) then
  if \( r_{l\text{in range}} = '1' \) then
    \text{start.final_range_check.state} <= 8;
    if \( \text{controller.shutdown.time} = 0 \) then
      if mem_updated /= last_mem_updated then -- there was an update! use it
        last_mem_updated <= mem_updated;
      end if;
    end if;
  end if;
else
  -- don't use the experimental controller until we're in the envelope for 5 secs
  \text{controller.shutdown.time} <= 165000000;
end if;
else if \( \text{start.final_range_check.state} = 9 \) then
  \text{range.limited.final.volts} <= \( r_{l\text{volt.out}} \);
  \text{start.final_range_check.state} <= 10;
  -- we're done with envelope, exit critical section
  \text{semaphore.checking.final.range} <= '0';
  end if;
end if;

------------- FINAL RANGE CHECK ABOVE ---------------

if \( \text{restart.receive} = '1' \) then
  \text{restart.receive} <= '0';
  \text{receive.state} <= \text{receive.track1};
else if \( \text{data.present} = '1' \) then
  -- we have received data which is in input_bits
  if \( \text{input_bits} = \text{x"ff}" \) then
    if \( \text{reset.state} = \text{got0} \) then
      \text{reset.state} <= \text{got1};
      else if \( \text{reset.state} = \text{got1} \) then
        \text{reset.state} <= \text{got2};
        else if \( \text{reset.state} = \text{got2} \) then
          \text{reset.state} <= \text{got3};
          else -- \( \text{reset.state} = \text{got4} \)
            \text{reset.state} <= \text{got5};  -- not really necessary since it will remain the same
          end if;
        else
          \text{reset.state} <= \text{got0};
        end if;
    case \text{receive.state} is
      when \text{receive.track1} =>
        \text{temp.receive.track} <= \text{x"000000"} & \text{input_bits};
        \text{receive.state} <= \text{receive.track2};
      when \text{receive.track2} =>
        \text{temp.receive.track} <= \text{x"0000"} & \text{input_bits} & \text{temp.receive.track}(7 downto 0);
        \text{receive.state} <= \text{receive.track3};
    end case;
  end if;
else
  \text{reset.state} <= \text{got0};
end if;

temp_received_track <= x"00" & input_bits & temp_received_track(15 downto 0);
receive_state <= receive_track4;

when receive_track4 =>
  -- if input_bits = 'x'0' then -- shouldn't happen, but you never know
  temp_received_track <= input_bits & temp_received_track(23 downto 0);
  --end if;
receive_state <= receive_angle1;

when receive_angle1 =>
temp_received_angle <= x"000000" & input_bits;
receive_state <= receive_angle2;

when receive_angle2 =>
temp_received_angle <= x"0000" & input_bits & temp_received_angle(7 downto 0);
receive_state <= receive_angle3;

when receive_angle3 =>
temp_received_angle <= x"00" & input_bits & temp_received_angle(15 downto 0);
receive_state <= receive_angle4;

when receive_angle4 =>
  received_angle <= input_bits & temp_received_angle(23 downto 0);
  received_track <= temp_received_track;
  store the values for the experimental controller
  start_reliable_controller_state <= 0;
receive_state <= reset fsm_start;
  -- start to send data (the result should be ready soon enough)
send_clock_count <= (others => '0');

when others =>
end case;
end if;
end if;
end process;

hr_controller_inst : hr_controller PORT MAP(
  start => hr_start,
clock => slow_clock,
angle => received_angle,
track => received_track,
cx_volts => hr_volts,
output_avail => hr_output_avail,

------------------- FPU signals
start_i => hr_start_i,
op_a_i => hr_opa_i,
op_b_i => hr_opb_i,
fp_op_i => hr_fp_op_i,
r_mode_i => hr_rmode_i,
output_o => fp_output_o,
ready_o => fp_ready_o);

rl_inst : range_limit PORT MAP(
  start => rl_start,
clock => slow_clock,
volt => rl_volt_in,
volt_out => rl_volt_out,
output_avail => rl_output_avail,
in_range => rl_in_range,
min_value => rl_range_min,
max_value => rl_range_max,

------------------- FPU signals
start_i => rl_start_i,
op_a_i => rl_opa_i,
op_b_i => rl_opb_i,
fp_op_i => rl_fp_op_i,
r_mode_i => rl_rmode_i,
A.2 Model Generator Code

The Model generator is split into five files. The two main ones, Main.java, and DiscreteBounded-MaxModel.java, are listed here, in that order.

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

/**
 * Main module for generating Maude model code
 * You can specify discretization constants for the various parameters we model. These are done by
 * giving a multiple of the base unit (which is the minimum possible discrete value). Base units are
 * Distance: millimeters
 * Angle: milliradians
 * Time: milliseconds
 * For example, if the discretization constant for acceleration is 10, then
 * each unit of velocity
 * counts as one centimeter per second, and each unit of acceleration is in
 * centimeters/second^2
 * @author Stan
 */
public class Main {
    private static final int UNDEFINED = -1999999999;

    // accel and vel
    private final static int FORWARDS_MIN_ACCEL_INDEX = 0;
    private final static int FORWARDS_MAX_ACCEL_INDEX = 1;
    private final static int BACKWARDS_MIN_ACCEL_INDEX = 2;
    private final static int BACKWARDS_MAX_ACCEL_INDEX = 3;
    private final static int MIN_VEL_INDEX = 4;
    private final static int MAX_VEL_INDEX = 5;
    private final static int BEST_RAW_ACCEL_INDEX = 6;
    private final static int BEST_OBSERVED_ACCEL_INDEX = 7;
    private final static int WORST_RAW_ACCEL_INDEX = 8;
    private final static int WORST_OBSERVED_ACCEL_INDEX = 9;
    private final static int TIME_FROM_BEST_TO_WORST_INDEX = 10;
    private final static int TIME_DISCRETIZATION_CONSTANT_INDEX = 11;
    private final static int DISTANCE_DISCRETIZATION_CONSTANT_INDEX = 12;
    private final static int ANGLE_DISCRETIZATION_CONSTANT_INDEX = 13;
    private final static int MAX_JUSTERING_ANGLE_INDEX = 14;
    private final static int MAX_JUSTERING_ANGLE_INDEX = 15;
```
private final static int DELTA_STEERING_ANGLE_INDEX = 16;
private final static int WHEEL_BASE_INDEX = 17;
private final static int TRACK_INDEX = 18;
private final static int COG_HEIGHT_INDEX = 19;
private final static int MAX_SLOPE_INDEX = 20;
private final static int MAX_DELTA_SLOPE_INDEX = 21;
private final static int SKID_BEST_RAW_ACCEL_INDEX = 22;
private final static int SKID_BEST_OBSERVED_ACCEL_INDEX = 23;
private final static int SKID_WORST_RAW_ACCEL_INDEX = 24;
private final static int SKID_WORST_OBSERVED_ACCEL_INDEX = 25;
private final static int SKID_TIME_FROM_BEST_TO_WORST_INDEX = 26;
private static final int NUM_INDEX = 27;

private static final String[] INDEX_NAMES = {

private static int[][] parameterValues = new int[NUM_INDEX][];

private static final String SLIP_DIR = "slip_model";
private static final String ROLL_DIR = "roll_model";
private static final String SKID_DIR = "skid_model";
String loadFile = ""; // the file to load from

if (args.length > 0) { loadFile = args[0]; } else { BufferedReader in = new BufferedReader(new InputStreamReader(System.in)); System.out.print("Please enter model txt filename: "); try { loadFile = in.readLine(); } catch (IOException e) { System.err.println(e); } }
loadValues(loadFile);
new File(SLIP_DIR).mkdirs();
new File(ROLL_DIR).mkdirs();
new File(SKID_DIR).mkdirs();
doAccel(SLIP_DIR);
doTerrainSlip(SLIP_DIR);
doVelocity(ROLL_DIR);
doSteering(ROLL_DIR);
doSlope(ROLL_DIR);
doVelocity(SKID_DIR);
doTerrainSkid(SKID_DIR);
doSteering(SKID_DIR);

System.out.println("Exiting...");
System.exit(0);
}

private static void loadValues(String filename)
{
    // sanity check
    if (NUMINDEX != INDEX_NAMES.length)
    {
        System.err.println("internal sanity check failed: NUMINDEX !=\nINDEX_NAMES.length");
        System.exit(1);
    }

    for (int x = 0; x < NUMINDEX; ++x)
        parameterValues[x] = UNDEFINED;

    if (filename =="")
    {
        System.out.println("Generating formal models from sample parameters...");

        parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] = 500; // in ms
        parameterValues[ANGLE_DISCRETIZATION_CONSTANT_INDEX] = 250; // in milliradians
        parameterValues[FORWARDS_MIN_ACCEL_INDEX] = -4000; // mm/s^2
        parameterValues[FORWARDS_MAX_ACCEL_INDEX] = 3000; // mm/s^2
        parameterValues[BACKWARDS_MIN_ACCEL_INDEX] = -2000; // mm/s^2
        parameterValues[BACKWARDS_MAX_ACCEL_INDEX] = 4000; // mm/s^2
        parameterValues[MIN_VEL_INDEX] = -4000; // mm/s
        parameterValues[MAX_VEL_INDEX] = 6000; // mm/s
        parameterValues[BEST_RAW_ACCEL_INDEX] = 4000; // mm/s^2
        parameterValues[BEST_OBSERVED_ACCEL_INDEX] = 3000; // mm/s^2
        parameterValues[WORST_RAW_ACCEL_INDEX] = 4000; // mm/s^2
        parameterValues[WORST_OBSERVED_ACCEL_INDEX] = 1000; // mm/s^2
        parameterValues[TIME_FROM_BEST_TO_WORST_INDEX] = 2000; // in ms
        parameterValues[MIN_STEERING_ANGLE_INDEX] = -1500; // in milliradians
        parameterValues[MAX_STEERING_ANGLE_INDEX] = 1500; // in milliradians
        parameterValues[DELTA_STEERING_ANGLE_INDEX] = 1000; // in milliradians per second
        parameterValues[WHEEL_BASE_INDEX] = 4000; // mm
        parameterValues[TRACK_INDEX] = 2000; // mm
        parameterValues[COG_HEIGHT_INDEX] = 1000; // mm
        parameterValues[MAX_SLOPE_INDEX] = 1000; // in milliradians, 1000 = 57 degrees
        parameterValues[MAX_DELTA_SLOPE_INDEX] = 250; // milliradians per second
        parameterValues[SKID_BEST_RAW_ACCEL_INDEX] = 4000; // mm/s^2
        parameterValues[SKID_BEST_OBSERVED_ACCEL_INDEX] = 3500; // mm/s^2
        parameterValues[SKID_WORST_RAW_ACCEL_INDEX] = 4000; // mm/s^2
        parameterValues[SKID_WORST_OBSERVED_ACCEL_INDEX] = 500; // mm/s^2
        parameterValues[SKID_TIME_FROM_BEST_TO_WORST_INDEX] = 2000; // in ms
    }
    else
    {
        // load them from filename
        try
        {
            BufferedReader in = new BufferedReader(\nnew InputStreamReader(new FileInputStream(filename)));

            for (String s = in.readLine(); s != null; s = in.readLine())
            {
                // first trim comments
                int commentIndex = s.indexOf(";\n");
                if (commentIndex != -1)
                    s = s.substring(0, commentIndex);
                s.trim();

                // ignore blank lines
                if (s.length() == 0)
                    continue;

            }
        }
        catch (Exception e)
        {
            System.out.println("Unable to load from file: ");
            System.out.println(filename);
        }
    }
}
String[] words = s.split("=");
if (words.length != 2)
{
    System.err.println("Warning: Model txt file line "+ s + " does not
contain exactly one equals sign "+");
    continue;
}
words[0] = words[0].trim().toLowerCase();
words[1] = words[1].trim().toLowerCase();
boolean found = false;
for (int x = 0; x < INDEX_NAMES.length; ++x)
{
    if (INDEX_NAMES[x].equals(words[0]))
    {
        parameterValues[x] = Integer.parseInt(words[1]);
        found = true;
        break;
    }
}
if (!found)
{
    System.out.println("Warning: Unknown parameter " + words[0] + " from
line "+ s + ";");
}
}
in.close();
}
catch (Exception e)
{
    System.err.println(e);
    System.exit(1);
}
}
for (int x = 0; x < NUM_INDEX; ++x)
if (parameterValues[x] == UNDEFINED)
{
    System.err.println("Error: Parameter " + INDEX_NAMES[x] + " was left
undefined.");
    System.exit(1);
}
}

public static void doSteering(String dir)
{
    String steeringFilename = dir + File.separator + "tractor−steering.maude";
    DiscreteBoundedMaxModel steerModel = new DiscreteBoundedMaxModel(null, "steerAngle");
    steerModel.setDiscretizationConstant(parameterValues[ANGLE_DISCRETIZATION_CONSTANT_INDEX]);
    steerModel.setTimeDiscretizationConstant(parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX]);
    steerModel.setIndependentRange(-90, 90);
    steerModel.setDependentRange(parameterValues[MAX_STEERING_ANGLE_INDEX], parameterValues[MAX_STEERING_ANGLE_INDEX]); // approximately [-PI/2, PI/2]
    steerModel.setPositiveIndependentRange(-parameterValues[DELTA_STEERING_ANGLE_INDEX], parameterValues[DELTA_STEERING_ANGLE_INDEX]);
    steerModel.setNegativeIndependentRange(-parameterValues[DELTA_STEERING_ANGLE_INDEX], parameterValues[DELTA_STEERING_ANGLE_INDEX]);
    PropertyModule steerModule = new PropertyModule("TRACTOR−STEERING");
    steerModule.addBodyGenerator(steerModel);
    // add reduce
    int deltaSteerAnglePerStep = parameterValues[DELTA_STEERING_ANGLE_INDEX] * parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] / 1000;
    steerModule.addBodyGenerator(new StringGenerator("op reduce−steerAngle : Int
−> steerAngle (cur−steerAngle + " +

50
steerModule.addBodyGenerator(new StringGenerator(" eq reduce-steerAngle (cur-steerAngle) = steerAngle (cur-steerAngle + 0 + deltaSteerAnglePerStep + ") if cur-steerAngle < 0 + deltaSteerAnglePerStep + 0.");
steerModule.addBodyGenerator(new StringGenerator(" eq reduce-steerAngle (cur-steerAngle) = steerAngle (0) [otherwise -0].");
steerModule.addBodyGenerator(new StringGenerator(" op initial-steerAngle:
steerAngle (0)."));
steerModule.addBodyGenerator(new StringGenerator(" op WHEEL-BASE:
WHEEL-BASE = (Float) (worstXIntercept * worstXIntercept)."));
steerModule.addBodyGenerator(new StringGenerator(" op TRACK:
TRACK = (Float) (parameterValues[TRACK_INDEX] + 0)."));
steerModule.addBodyGenerator(new StringGenerator(" op COG-HEIGHT:
COG-HEIGHT = parameterValues[COG_HEIGHT_INDEX]."));
try {
PrintStream ps = new PrintStream(steeringFilename);
steerModule.generate(ps);
ps.close();
} catch (Exception e) {
System.out.println(e);
}
public static void doVelocity(String directory) {
String accelFilename = directory + File.separator + "tractor-vel.maud:\
if (worstXIntercept == -1) {
System.err.println("Error: doTerrain must be done before doVelocity (need worstXIntercept)");
System.exit(1);
}
float forwardsSlipBreaking = (parameterValues[FORWARDS_MIN_ACCEL_INDEX] * parameterValues[FORWARDS_MIN_ACCEL_INDEX]) / (float) (worstXIntercept * worstXIntercept);
float backwardsSlipBreaking = (parameterValues[BACKWARDS_MAX_ACCEL_INDEX] * parameterValues[BACKWARDS_MAX_ACCEL_INDEX]) / (float) (worstXIntercept * worstXIntercept);
int forwardsBreakingAccel = (int) (parameterValues[FORWARDS_MIN_ACCEL_INDEX] * (1.0 - forwardsSlipBreaking));
int backwardsBreakingAccel = (int) (parameterValues[BACKWARDS_MAX_ACCEL_INDEX] * (1.0 - backwardsSlipBreaking));
if (parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] > 1000) {
System.err.println("error: time discretization constant < 1000 (can't compute steps per second).\nSystem.exit(1);
}
int stepsPerSecond = 1000 / parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX];
System.out.println("stepsPerSecond = " + stepsPerSecond);
for (; (forwardsBreakingAccel / stepsPerSecond) % parameterValues[DISTANCE_DISCRETIZATION_CONSTANT_INDEX] != 0 ||
forwardsBreakingAccel % stepsPerSecond != 0) {
++forwardsBreakingAccel;
}
for (; (backwardsBreakingAccel / stepsPerSecond) %
parameterValues[DISTANCE_DISCRETIZATION_CONSTANT_INDEX] != 0 ||
backwardsBreakingAccel % stepsPerSecond != 0; )
−−
backwardsBreakingAccel;

int ideal_forwards_breaking_accel = (int)((1.0 − forwardsSlipBreaking) *
parameterValues[FORWARDS_MIN_ACCEL_INDEX]);
int ideal_backwards_breaking_accel = (int)((1.0 − backwardsSlipBreaking) *
parameterValues[BACKWARDS_MAX_ACCEL_INDEX]);

System.out.println("INFO(pessimism): "+(100.0 −
100.0 *
forwardsBreakingAccel / ideal_forwards_breaking_accel) + "% change in worst case forward breaking in " +
" + ideal_forwards_breaking_accel + " -> " + forwardsBreakingAccel + ");

System.out.println("INFO(pessimism): "+(100.0 −
100.0 *
backwardsBreakingAccel / ideal_backwards_breaking_accel) + "% change in worst case backwards breaking in " +
" + ideal_backwards_breaking_accel + " -> " + backwardsBreakingAccel + ");

DiscreteBoundedMaxModel velModel = new DiscreteBoundedMaxModel(null, "vel");

velModel.setDiscretizationConstant(parameterValues[DISTANCE_DISCRETIZATION_CONSTANT_INDEX]);
velModel.setTimeDiscretizationConstant(parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX]);
velModel.setPositiveIndependentRange(forwardsBreakingAccel, parameterValues[FORWARDS_MAX_ACCEL_INDEX]); // in mm/s^2
velModel.setNegativeIndependentRange(parameterValues[BACKWARDS_MIN_ACCEL_INDEX],
backwardsBreakingAccel); // in mm/s^2
velModel.setDependentRange(parameterValues[MIN_VEL_INDEX],
parameterValues[MAX_VEL_INDEX]); // in mm/s

PropertyModule velModule = new PropertyModule("TRACTOR−VEL");
velModule.addBodyGenerator(velModel);

// add reduce
velModule.addBodyGenerator(new StringGenerator("op reduce−vel : Int ->
SORT_VEL . "));
velModule.addBodyGenerator(new StringGenerator("ceq reduce−vel(cur−vel) =
vel(cur−vel + " + forwardsBreakingAccel / stepsPerSecond + ") if cur−vel > 
+ −(forwardsBreakingAccel / stepsPerSecond + ");
velModule.addBodyGenerator(new StringGenerator("ceq reduce−vel(cur−vel) =
vel(cur−vel + " + backwardsBreakingAccel / stepsPerSecond + ") if cur−vel < 
+ −(backwardsBreakingAccel / stepsPerSecond + ");
velModule.addBodyGenerator(new StringGenerator("eq reduce−vel(cur−vel) =
vel(0) [otherwise] ");
velModule.addBodyGenerator(new StringGenerator(" ");
velModule.addBodyGenerator(new StringGenerator("op initial−vel : ->
SORT2_STATE . "));
velModule.addBodyGenerator(new StringGenerator("eq initial−vel = vel(0) . "));
velModule.addBodyGenerator(new StringGenerator(" "));
velModule.addBodyGenerator(new StringGenerator("op VEL_INCREMENT : -> Int . "));
velModule.addBodyGenerator(new StringGenerator("eq VEL_INCREMENT = " +
parameterValues[DISTANCE_DISCRETIZATION_CONSTANT_INDEX] + " . "));

try {
PrintStream ps = new PrintStream(accelFilename);
velModule.generate(ps);
ps.close();
}
catch (Exception e) {
System.out.println(e);
}

public static void doSlope(String directory) {
String slopeFilename = directory + File.separator + "tractor−slope.maude";
DiscreteBoundedMaxModel slopeModel = new DiscreteBoundedMaxModel(null, "slope");
slopeModel.setDiscretizationConstant(parameterValues[ANGLE_DISCRETIZATION_CONSTANT_INDEX]);
}
slopeModel.setTimeDiscretizationConstant(parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX]);
slopeModel.setPositiveIndependentRange(-parameterValues[MAXDELTAJLOPINDEX],
parameterValues[MAXDELTAJLOPINDEX]); // in mm/s^2
slopeModel.setNegativeIndependentRange(-parameterValues[MAXDELTAJLOPINDEX],
parameterValues[MAXDELTAJLOPINDEX]); // in mm/s^2
slopeModel.setDependentRange(-parameterValues[MAXJLOPINDEX],
parameterValues[MAXJLOPINDEX]); // in mm/s

PropertyModule slopeModule = new PropertyModule("TRACTOR-SLOPE");
slopeModule.addBodyGenerator(slopeModel);

int slopePerTick = parameterValues[MAXDELTAJINDEX] *
parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX] / 1000;

slopeModule.addBodyGenerator(new StringGenerator("--- max slope increment"));
slopeModule.addBodyGenerator(new StringGenerator("op increase-slope : Int ->
SORT\_SLOPE "));
slopeModule.addBodyGenerator(new StringGenerator("eq increase-slope (cur-slope) = slope (trim-slope (cur-slope + " + slopePerTick + ")"));
slopeModule.addBodyGenerator(new StringGenerator("op initial-slope :
SORT\_STATE "));
slopeModule.addBodyGenerator(new StringGenerator("eq initial-slope = slope (0) ",
" "));

try {
    PrintStream ps = new PrintStream(slopeFilename);
    slopeModule.generate(ps);
    ps.close();
}

catch (Exception e) {
    System.out.println(e);
}
}

public static void doAccel(String directory)
{
    String accelFilename = directory + File.separator + "tractor-accel.maude";

    DiscreteBoundedMaxModel accelModel = new DiscreteBoundedMaxModel("raw-accel",
"raw-vel");

    accelModel.setTimeDiscretizationConstant(parameterValues[DISTANCEDISCRETIZATION_CONSTANT_INDEX]); // mm
    accelModel.setTimeDiscretizationConstant(parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX]); // ms

    accelModel.setPositiveIndependentRange(parameterValues[FORWARDSMINACCELINDEX],
parameterValues[FORWARDSMAXACCELINDEX]); // in mm/s^2
    accelModel.setNegativeIndependentRange(parameterValues[BACKWARDSMINACCELINDEX],
parameterValues[BACKWARDSMAXACCELINDEX]); // in mm/s^2
    accelModel.setDependentRange(parameterValues[MINVELINDEX],
parameterValues[MAXVELINDEX]); // in mm/s

    PropertyModule accelModule = new PropertyModule("TRACTOR-ACCEL");
    accelModule.addBodyGenerator(accelModel);

    if ((parameterValues[FORWARDSMINACCELINDEX] *
parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX]) % 1000 != 0)
    {
        System.err.println("FORWARDSMINACCELINDEX( +
parameterValues[FORWARDSMINACCELINDEX] + ") *
TIMEDISCRETIZATION_CONSTANT_INDEX( +
parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX] + ") is not divisible
by 1000");
        System.exit(1);
    }

    if ((parameterValues[BACKWARDSMAXACCELINDEX] *
parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX]) % 1000 != 0)
    {
        System.err.println("BACKWARDSMAXACCELINDEX( +
parameterValues[BACKWARDSMAXACCELINDEX] + ") *
TIMEDISCRETIZATION_CONSTANT_INDEX( +
parameterValues[TIMEDISCRETIZATION_CONSTANT_INDEX] + ") is not divisible
by 1000);  
  System. exit (1);  
}  
  // add reduce  
  int forwardReduceStep = parameterValues [FORWARDS_MAX_ACCEL_INDEX] *  
  parameterValues [TIME_DISCRETIZATION_CONSTANT_INDEX] / 1000;  
  int backwardsReduceStep = parameterValues [BACKWARDS_MAX_ACCEL_INDEX] *  
  parameterValues [TIME_DISCRETIZATION_CONSTANT_INDEX] / 1000;  
  accelModule. addBodyGenerator(new StringGenerator (" vars cur−accel: Int.");  
  accelModule. addBodyGenerator(new StringGenerator (" op reduce−accel: Int.");  
  accelModule. addBodyGenerator(new StringGenerator (" ceq reduce−accel(cur−accel) = cur−accel + (" + forwardReduceStep + "+") if cur−accel > " + forwardReduceStep + ".");  
  accelModule. addBodyGenerator(new StringGenerator (" ceq reduce−accel(cur−accel) = cur−accel + (" + backwardsReduceStep + "+") if cur−accel < " + backwardsReduceStep + ".");  
  accelModule. addBodyGenerator(new StringGenerator (" eq reduce−accel(cur−accel) = 0 [swi ese, ] .");  
  accelModule. addBodyGenerator(new StringGenerator (" op initial−accel: Int.");  
  accelModule. addBodyGenerator(new StringGenerator (" eq initial−accel = advance−raw−accel(0 raw−vel(0)) raw−vel(0) .");  
  accelModule. addBodyGenerator(new StringGenerator (" op VEL_INCREMENT: Int.");  
  accelModule. addBodyGenerator(new StringGenerator (" eq VEL_INCREMENT = " + parameterValues [DISTANCE_DISCRETIZATION_CONSTANT_INDEX] + ".");  
  try  
  {  
  PrintStream ps = new PrintStream (accelFilename);  
  accelModule. generate (ps);  
  ps. close();  
  }  
  catch ( Exception e)  
  {  
  System. out. printin (e);  
  }  
}  
/*  parameterValues [SKID_BEST_VEL_IDEAL_INDEX] = 0; // mm/s  parameterValues [SKID_BEST_WHEEL_ANGLE_INDEX] = 0; // milliradians  parameterValues [SKID_BEST_RADIUS_INDEX] = 0; //mm  parameterValues [SKID_BEST_TIME_INDEX] = 0; // ms  parameterValues [SKID_WORST_VEL_IDEAL_INDEX] = 0; // mm/s  parameterValues [SKID_WORST_WHEEL_ANGLE_INDEX] = 0; // milliradians  parameterValues [SKID_WORST_RADIUS_INDEX] = 0; //mm  parameterValues [SKID_WORST_TIME_INDEX] = 0; // ms  */  
public static void doTerrainSkid (String directory)  
{  
  String accelFilename = directory + File. separator +  
  "tractor−skidterrain. maude";  
  int best_raw_accel = parameterValues [SKID_BEST_RAW_ACCEL_INDEX];  
  int best_observed_accel = parameterValues [SKID_BEST_OBSERVED_ACCEL_INDEX];  
  int worst_raw_accel = parameterValues [SKID_WORST_RAW_ACCEL_INDEX];  
  int worst_observed_accel = parameterValues [SKID_WORST_OBSERVED_ACCEL_INDEX];  
  int time_from_best_to_worst = parameterValues [SKID_TIME_FROM_BEST_TO_WORST_INDEX];  
  if (best_raw_accel <= best_observed_accel)  
  {  
  System.err. printin ("Error: Skid Terrain Best raw accel(" + best_raw_accel + ") must be more than best observed accel(" + best_observed_accel + ")");  
  }  
  }
```java
System.exit(0);
}
if (worst_raw_accel <= worst_observed_accel)
{
    System.err.println("Error: Skid Terrain Worst raw accel(" + worst_raw_accel + ") must be more than worst observed accel(" + worst_observed_accel + ")");
    System.exit(8);
}
int best_slip = 1000 - (1000 * best_observed_accel) / best_raw_accel;
int best_x_intercept = (int) Math.round(best_raw_accel / Math.sqrt(best_slip / 1000.0));
int worst_slip = 1000 - (1000 * worst_observed_accel) / worst_raw_accel;
worst_x_intercept = (int) Math.round(worst_raw_accel / Math.sqrt(worst_slip / 1000.0));

int initial_worst_x_intercept = worst_x_intercept;
if (time_from_best_to_worst % parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] != 0)
{
    System.err.println("Error: Skid time from best to worst(" + 
    time_from_best_to_worst + ") is not a multiple of the time discretization constant(" 
    + parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] + ")");
    System.exit(1);
}
int step_count_from_best_to_worst = time_from_best_to_worst / parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX];
// System.out.println("steps = " + step_count_from_best_to_worst);
int x_intercept_diff = best_x_intercept - worst_x_intercept;
int x_intercept_step = x_intercept_diff / step_count_from_best_to_worst;

// here we want to decrease worst_x_intercept such that best_x_intercept - worst_x_intercept
// is exactly divisible by step_count_from_best_to_worst and
// x_intercept_diff = timeDiscretizationConstant % 1000 = 0

// first increase x_intercept_diff if necessary
// to make sure (x_intercept_diff = timeDiscretizationConstant % 1000 = 0)
for (;
    timeDiscretizationConstant % 1000 != 0 ||
    (best_x_intercept - worst_x_intercept) % step_count_from_best_to_worst
    != 0;
)
{
    --worst_x_intercept;
    // worst_x_intercept = worst_x_intercept - step_count_from_best_to_worst;
    --worst_x_intercept;
    ++x_intercept_diff;
    x_intercept_step = x_intercept_diff / step_count_from_best_to_worst;
}
// System.out.println("x_intercept_diff = " + x_intercept_diff);
// System.out.println("step_count_from_best_to_worst = " + step_count_from_best_to_worst);

System.out.println("INFO( pessimism): subtracted " + (100 *
    (initial_worst_x_intercept - worst_x_intercept)) / (float) initial_worst_x_intercept + 
    "% from worst-case x_intercept in " + directory + " (+
    initial_worst_x_intercept + " - > " + 
    worst_x_intercept + ")");

DiscreteBoundedMaxModel xIntModel = new DiscreteBoundedMaxModel(null, 
    "x_intercept");
// System.out.println("disc constant = " + x_intercept_step);
xIntModel.setDiscretizationConstant(x_intercept_step);
xIntModel.setTimeDiscretizationConstant(parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX]);
xIntModel.setPositiveIndependentRange(-x_intercept_diff * 1000 / 
    time_from_best_to_worst, x_intercept_diff * 1000 / time_from_best_to_worst); // in xInt1000/s
xIntModel.setDependentRange(worst_x_intercept, best_x_intercept); // in xInt1000/s
PropertyModule xIntModule = new PropertyModule("TRACTOR-TerrAIN");
xIntModule.addBodyGenerator(xIntModel);
xIntModule.addBodyGenerator(new StringGenerator("op initial-x_intercept : 
    - > SORTSTATE . ")};
```
try {
    PrintStream ps = new PrintStream(accelFilename);
    x_int_Module.generate(ps);
    ps.close();
} catch (Exception e) {
    System.out.println(e);
}

public static void doTerrainSlip(String directory) {
    String accelFilename = directory + File.separator + "tractor-slip terrain.maude";
    int best_raw_accel = parameterValues[BEST_RAW_ACCEL_INDEX];
    int best_observed_accel = parameterValues[BEST_OBSERVED_ACCEL_INDEX];
    int worst_raw_accel = parameterValues[WRST_RAW_ACCEL_INDEX];
    int worst_observed_accel = parameterValues[WRST_OBSERVED_ACCEL_INDEX];
    int time_from_best_to_worst = parameterValues[TIME_FROM_BEST_TO_WORST_INDEX];
    if (best_raw_accel <= best_observed_accel) {
        System.err.println("Error: Slip Terrain Best raw accel(" + best_raw_accel + ") must be more than best observed accel(" + best_observed_accel + ");");
        System.exit(0);
    }
    if (worst_raw_accel <= worst_observed_accel) {
        System.err.println("Error: Slip Terrain Worst raw accel(" + worst_raw_accel + ") must be more than worst observed accel(" + worst_observed_accel + ");");
        System.exit(0);
    }
    int best_slip = 1000 - (1000 * best_observed_accel) / best_raw_accel;
    int best_x_intercept = (int)Math.round(best_raw_accel / Math.sqrt(best_slip / 1000.0));
    int worst_slip = 1000 - (1000 * worst_observed_accel) / worst_raw_accel;
    worst_x_intercept = (int)Math.round(worst_raw_accel / Math.sqrt(worst_slip / 1000.0));
    int initial_worst_x_intercept = worst_x_intercept;
    if (time_from_best_to_worst %
        parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] != 0) {
        System.err.println("Error: Slip time from best to worst(" +
            time_from_best_to_worst + ") is not a multiple of the time discretization constant(" +
            parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] + ");");
        System.exit(1);
    }
    int step_count_from_best_to_worst = time_from_best_to_worst /
        parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX];
    //System.out.println("steps = " + step_count_from_best_to_worst);
    int x_intercept_diff = best_x_intercept - worst_x_intercept;
    int x_intercept_step = x_intercept_diff / step_count_from_best_to_worst;
import java.io.PrintStream;

// here we want to decrease worst.x_intercept such that best.x_intercept - worst.x_intercept
// is exactly divisible by step_count_from_best_to_worst and
// x_intercept_diff * timeDiscretizationConstant % 1000 == 0

// first increase x_intercept_diff if necessary
// to make sure (x_intercept_diff * TIME_DISCRETIZATION_CONSTANT) % 1000 == 0
for (;;) {
    x_intercept_diff = (x_intercept_diff * timeDiscretizationConstant) % 1000 == 0
    if (best.x_intercept - worst.x_intercept) % step_count_from_best_to_worst
        x_intercept += worst.x_intercept;
    ++x_intercept_step = (x_intercept_diff / step_count_from_best_to_worst);
}

System.out.println("x_intercept_diff = " + x_intercept_diff);
System.out.println("step_count_from_best_to_worst = " +
    step_count_from_best_to_worst);

DiscreteBoundedMaxModel x_int_Model = new DiscreteBoundedMaxModel(null,
    "x_intercept");

// System.out.println("disc constant = " + x_intercept_step);
x_int_Model.setTimeDiscretizationConstant(x_intercept_step);

x_int_Model.setPositiveIndependentRange(-x_intercept_diff / 1000 /
    time_from_best_to_worst, x_intercept_diff * 1000 / time_from_best_to_worst);

// in x_int*1000/

x_int_Model.setDependentRange(worst.x_intercept, best.x_intercept);

int stepDif = parameterValues[TIME_DISCRETIZATION_CONSTANT_INDEX] *
    x_intercept_diff * 1000 / (time_from_best_to_worst * 1000);

PropertyModule x_int_Module = new PropertyModule("TRACTOR-TERRAIN");

x_int_Module.addBodyGenerator(new StringGenerator("op initial-x_intercept :
    -> SORT-STATE ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq initial-x_intercept = x_intercept + " +
    initial_x_intercept + "' ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq initial-x_intercept = x_intercept + " +
    x_intercept_step + "' ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq decreaseXInt( +) : Int
    -> Int " ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq decreaseXInt(cur-x_intercept = cur-x_intercept - " +
    stepDif + " if
cur-x_intercept > " + worst_x_intercept + " ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq decreaseXInt(cur-x_intercept = cur-x_intercept + " +
    worst_x_intercept + " ");

x_int_Module.addBodyGenerator(new StringGenerator("'eq XINT-INCREMENT : -> Int ");

x_int_Module.addBodyGenerator(new StringGenerator("op XINT-INCREMENT = " +
    stepDif + ");

try {
    PrintStream ps = new PrintStream(accelFilename);
    x_int_Module.generate(ps);
    ps.close();
} catch (Exception e) {
    System.out.println("e");
}
}
This models where we have one variable depending on another like velocity depends on acceleration.

Both variables are bounded with upper and lower bounds.
The independent variable has a range that it can vary in when the dependent one is positive.
and a different range for when the dependent value is negative.

For example, the possible acceleration changes are different when the tractor is in reverse versus
when it's moving forward (since breaking is stronger than thrusting?).

@Author Stan

```java
public class DiscreteBoundedMaxModel extends Generatable {
    private int discretizationConstant = 1;
    private int timeDiscretizationConstant = 1;
    private boolean dividedRanges = false; // have we divided the ranges by the
time discretization constant?

    private String independentValue = null;
    private String dependentValue = null;

    private int[] dependentRange = new int[2];
    private int[] positiveIndependentRange = new int[2];
    private int[] negativeIndependentRange = new int[2];

    public DiscreteBoundedMaxModel(String independentValue, String dependentValue) {
        this.independentValue = independentValue;
        this.dependentValue = dependentValue;
    }

    public void setDiscretizationConstant(int discretizationConstant) {
        this.discretizationConstant = discretizationConstant;
    }

    public void setTimeDiscretizationConstant(int timeDiscretizationConstant) {
        this.timeDiscretizationConstant = timeDiscretizationConstant;
    }

    public void setDependentRange(int min, int max) {
        dependentRange[0] = min;
        dependentRange[1] = max;
    }

    public void setPositiveIndependentRange(int min, int max) {
        positiveIndependentRange[0] = min;
        positiveIndependentRange[1] = max;
    }

    public void setNegativeIndependentRange(int min, int max) {
        negativeIndependentRange[0] = min;
        negativeIndependentRange[1] = max;
    }

    public void generateDependentModule(PrintStream out) throws Exception {
        // this module only has the dependent variable in the module (less state)

        String sortDependent = "SORT" + dependentValue.toUpperCase();
        out.println("sort " + sortDependent + " .");
        out.println();
        out.println("subsort " + sortDependent + " < STATE .");
        out.println();

        //out.println("op trim−" + dependentValue + "(\_) : Int −> Int .");
        out.println("op trim−" + dependentValue + "(\_) : Int −> Int .");
        out.println("op advance−" + dependentValue + "(\_) : Int −> " + sortDependent + " .");
        out.println("op " + dependentValue + "(\_) : Int −> " + sortDependent + " .");
        out.println();
    }
}
```
```java
println(" vars cur~" + dependentValue + " : Int . ");
println();
println("--- advance " + dependentValue);

if ( dividedRanges == false)
{
  checkRanges();

  negativeIndependentRange[0] = timeDiscretizationConstant * negativeIndependentRange[0] / 1000;
  positiveIndependentRange[0] = timeDiscretizationConstant * positiveIndependentRange[0] / 1000;
  dividedRanges = true;
}

int min = Math.min(negativeIndependentRange[0], positiveIndependentRange[0]);
int max = Math.max(negativeIndependentRange[1], positiveIndependentRange[1]);

// System.out.println(dependentValue + " range = " + min + ", " + max + ";
stepsPerSecond = " + stepsPerSecond);
for (int x = max; x >= min; x -= discretizationConstant)
{
  int value = x;
  // case 1: we're in negative range but not in positive range
  if (x >= negativeIndependentRange[0] && x <= negativeIndependentRange[1] &&
      x < positiveIndependentRange[0] || x > positiveIndependentRange[1])
  {
    println(" crl advance~" + dependentValue + "+ (cur~" + dependentValue + "+")
            + " + dependentValue + "(trim~" + dependentValue + "+" + value + 
            ") if cur~" + dependentValue + " <= " + (-value) + " . ");
  }

  // case 2: we're in positive range but not in negative range
  else if (x >= positiveIndependentRange[0] && x <=
             positiveIndependentRange[1] &&
             x < negativeIndependentRange[0] || x > negativeIndependentRange[1])
  {
    println(" crl advance~" + dependentValue + "+ (cur~" + dependentValue + "+")
            + " + dependentValue + "(trim~" + dependentValue + "+" + value + 
            ") if cur~" + dependentValue + " >= " + (-value) + ");
  }

  // case 3: we're in range for both
  else
  {
    println(" r1 advance~" + dependentValue + "+ (cur~" + dependentValue + "+")
            + " + dependentValue + "(trim~" + dependentValue + "+" + value + 
            ")");
  }
}

// if they defined a dependent range
if (dependentRange[0] != dependentRange[1])
{
  println();
  println("req trim~" + dependentValue + "(cur~" + dependentValue + "+") = "
            + dependentRange[1] + " if cur~" + dependentValue + "> " +
            dependentRange[1] + " . ");
  println("req trim~" + dependentValue + "(cur~" + dependentValue + "+") = "
            + dependentRange[0] + " if cur~" + dependentValue + "> " +
            dependentRange[0] + " . ");
  println("eq trim~" + dependentValue + "(cur~" + dependentValue + "+") = "
            + dependentValue + " [otherwise] ");
  println();
}

private void checkRanges() throws Exception
{
  int min = Math.min(negativeIndependentRange[0], positiveIndependentRange[0]);
  int max = Math.max(negativeIndependentRange[1], positiveIndependentRange[1]);
  float stepsPerSecond = 1000/timeDiscretizationConstant;
}
```

// generate rules
if ( (max - min) \% discretizationConstant != 0)
{
    throw new Exception(" Range [" + min + ", " + max + "] is not divisible by discretizationConstant [" + discretizationConstant + "]");
}

if (Math.round((positiveIndependentRange[1] - positiveIndependentRange[0]) / stepsPerSecond) \% discretizationConstant != 0)
{
}

if (Math.round((negativeIndependentRange[1] - negativeIndependentRange[0]) / stepsPerSecond) \% discretizationConstant != 0)
{
    throw new Exception(" Negative Independent Range [" + negativeIndependentRange[0] + ", " + negativeIndependentRange[1] + "] / stepsPerSecond [" + stepsPerSecond + "] = " + Math.round((negativeIndependentRange[1] - negativeIndependentRange[0]) / stepsPerSecond) + " is not divisible by discretizationConstant [" + discretizationConstant + "]");
}

if (Math.round((dependentRange[1] - dependentRange[0]) / stepsPerSecond) \% discretizationConstant != 0)
{
    throw new Exception(" Range [" + dependentRange[1] + ", " + dependentRange[0] + "] / stepsPerSecond [" + stepsPerSecond + "] = " + Math.round((dependentRange[1] - dependentRange[0]) / stepsPerSecond) + " is not divisible by discretizationConstant [" + discretizationConstant + "]");
}

public void generateIndependentDependentModule(PrintStream out) throws Exception
{
    String sortIndependent = "SORT" + independentValue.toUpperCase();
    String sortDependent = "SORT" + dependentValue.toUpperCase();
    out.println(" sorts " + sortIndependent + " " + sortDependent + " . ");
    out.println(" subsort " + sortIndependent + " " + sortDependent + "; SORT
STATE . ");
    out.println(" ops trim " + independentValue + "() : Int -> Int . ");
    out.println(" ops advance " + independentValue + "() : Int -> Int . ");
    out.println(" vars cur " + dependentValue + " cur " + independentValue + " : Int . ");
    out.println("-- advance " + independentValue + " . ");
    // modify by time disc value
    if (dividedRanges == false)
    {
        checkRanges();
        negativeIndependentRange[0] = timeDiscretizationConstant * negativeIndependentRange[0] / 1000;
    }
}
positiveIndependentRange[0] = timeDiscretizationConstant * positiveIndependentRange[0] / 1000;
dividedRanges = true;
}
// generative rules
int min = Math.min( negativeIndependentRange[0] , positiveIndependentRange[0] );
int max = Math.max( negativeIndependentRange[1] , positiveIndependentRange[1] );
/*
 System.out.println(" sort = " + sortDependent );
 System.out.println(" min = " + min + " , max = " + max + "; timeDiscConstant = " + timeDiscretizationConstant );
*/
if ( (max - min) % discretizationConstant != 0 )
{ throw new Exception( " Range [ " + min + " , " + max + "] is not divisible by discretizationConstant (" + discretizationConstant + ")" ) ; }
if ( (positiveIndependentRange[1] - positiveIndependentRange[0]) % discretizationConstant != 0 )
{ throw new Exception( " Positive Independent Range [ " + positiveIndependentRange[0] + " , " + positiveIndependentRange[1] + "] is not divisible by discretizationConstant (" + discretizationConstant + ")" ) ; }
if ( (negativeIndependentRange[1] - negativeIndependentRange[0]) % discretizationConstant != 0 )
{ throw new Exception( " Negative Independent Range [ " + negativeIndependentRange[0] + " , " + negativeIndependentRange[1] + "] is not divisible by discretizationConstant (" + discretizationConstant + ")" ) ; }
if ( (dependentRange[1] - dependentRange[0]) % discretizationConstant != 0 )
{ throw new Exception( " Range [ " + dependentRange[0] + " , " + dependentRange[1] + "] is not divisible by discretizationConstant (" + discretizationConstant + ")" ) ; }
for ( int x = max ; x >= min ; x -= discretizationConstant )
{
 // r l advance-raw-accel( cur-raw-accel raw-vel( cur-raw-vel ) ) =
 raw-accel(250) .
  // case 1: we're in negative range but not in positive range
  if ( x >= negativeIndependentRange[0] && x <= negativeIndependentRange[1] &&
( x < positiveIndependentRange[0] || x > positiveIndependentRange[1] ) )
{ if ( (-x) < dependentRange[0] ) // out of range continue;
  out.println( " r l advance-" +
 independentValue + "( cur-" + independentValue + " + dependentValue +
 (cur-" + dependentValue + ") ) = " +
 dependentValue + "(" + x + ") if cur-" +
 dependentValue + " <= " + ( -x ) + " ." );
}
  // case 2: we're in positive range but not in negative range
  else if ( x >= positiveIndependentRange[0] && x <=
 positiveIndependentRange[1] &&
( x < negativeIndependentRange[0] || x > negativeIndependentRange[1] ) )
{ if ( (-x) > dependentRange[1] ) // out of range continue;
  out.println( " r l advance-" +

A.3 Sample Model File

Here is a sample model file, which can be input into our model generator to produce a formal Maude model of the system.

```plaintext
; This is an example of an input model for the Maude Formal Model Generator
; Designed in collaboration with John Deere
; created by Stan Bak (sbaok2@illinois.edu), 3/2009

TimeDiscretizationConstant = 500
DistanceDiscretizationConstant = 250
```

---

This is an example of an input model for the Maude Formal Model Generator.
The discrete unit of angle measurement, in milliradians
AngleDiscretizationConstant = 200

These define the physical dimensions of the tractor. The measurements for the
wheel base, track, and height of the center of gravity are in millimeters
WheelBase = 4000
Track = 2000
CogHeight = 1000

These define the potential behavior of the tractor when the velocity is
positive.
1. Since breaking is typically more intense than accelerating, expect the min
   accel here to
2. have a higher absolute value. Expressed in millimeters/second^2
ForwardsMinAccel = −3000
ForwardsMaxAccel = 2000

Same as above, except when the velocity is negative.
BackwardsMinAccel = −1000
BackwardsMaxAccel = 3000

These are the absolute limits of the velocity of the tractor. Expressed in
millimeters/second
MinVel = −1000
MaxVel = 6500

These define the behavior of the steering of the tractor. The angles are
expressed in
milliradians, and we define a minimum and maximum angle. 1.5 radians ≈ 86
degrees.
1. Delta steering angle is maximum change in steering per second
   (milliradians/second)
MinSteeringAngle = −1400
MaxSteeringAngle = 1400
DeltaSteeringAngle = 800

These characterize the terrain. MaxSlope is the maximum tilt of the terrain,
in milliradians.
1 radian ≈ 57 degrees. Max delta slope is the maximum change in slope at the
maximum velocity of the tractor. This is expressed in milliradians per second
MaxSlope = 1000
MaxDeltaSlope = 400

The next values measure the friction of various surfaces where our tractor
will operate.
Two measurements are required, one on the least slippery surface, and one on
the most
1. slippery surface (the ones between will be inferred). Raw or ideal accel
   refers to the
2. acceleration without slip (the one reported on the CAN bus). Observed
   acceleration is
3. acceleration after slip. Best is for the least slippery surface, worst means
   most
   slippery surface
BestRawAccel = 1000
BestObservedAccel = 750
WorstRawAccel = 1000
WorstObservedAccel = 250

This is the time in milliseconds that it takes to drive from the least
slippery surface to
1. the most slippery surface. Since this is just a performance function, it's
2. okay to be
3. slightly off. A larger value will be less pessimistic at the expense of
   potential later
detection.
TimeFromBestToWorst = 2000

These are the same as above, except they are used for the skid (oversteering)
model. The
1. acceleration here is referring to the acceleration of the turn, which can be
   inferred by
2. performing a turn and measuring the difference between the expected radius
   and the observed
3. radius. There is likely to be less slip than the slip model (which you input
   above)
SkidBestRawAccel = 4000
SkidBestObservedAccel = 3500
SkidWorstRawAccel = 4000
SkidWorstObservedAccel = 500
SkidTimeFromBestToWorst = 2000
A.4 VMaude Semantics

This Maude file defines the semantics for all the implemented VMaude statements.

--- Stanley Bak
--- sbak2@illinois.edu
--- 5/2009
--- VMaude Semantics Definition File
---
--- notes:
--- you must have (cycle-count) in your list of VMaude program
--- you must define a signal named 'fsmState'

mod VHDL-STRUCTURED is
  pr INT .
  pr QID .
  sort SORT_VHDL_LINE
  sort VARIABLE_VALUE
  sort SORT_INITIAL_VALUE .

op noline : -> SORT_VHDL_LINE .
op noinput : -> SORT_INITIAL_VALUE .

vars VarValue IntValue Length OtherLength OldInt StateNum NextStateNum TrueNextStateNum FalseNextStateNum Count : Int .
vars VarName OtherVarName ThirdVarName : Qid .
vars Lines : SORT_VHDL_LINE .
vars OtherInitializers : SORT_INITIAL_VALUE .

op (input __) : Qid Int -> SORT_VHDL_INPUT .
op input-value (__ ) : Qid Int Int -> SORT_VHDL_INPUT .
eq (input VarName Length) = input-value (VarName Length 0) .

op (output __) : Qid Int -> SORT_VHDL_OUTPUT .
op output-value (__ ) : Qid Int Int -> SORT_VHDL_OUTPUT .
eq (output VarName Length) = output-value (VarName Length 0) .

op (signal __) : Qid Int -> SORT_VHDL_SIGNAL .
op signal-value (__ ) : Qid Int Int -> SORT_VHDL_SIGNAL .
eq (signal VarName Length) = signal-value (VarName Length 0) .

op (cycle-count) : -> SORT_VHDL_LINE .
op cyclecount (__ ) : Int -> SORT_VHDL_LINE .
eq (cycle-count) = cyclecount (0) .

--- extract a value from within the module
op getInternalValue (__, __) : Qid SORT_VHDL_LINE -> Int
  eq getInternalValue (VarName , Lines input-value (VarName Length VarValue) ) = VarValue .
  eq getInternalValue (VarName , Lines signal-value (fsmState Length VarValue) ) = VarValue .

--- get a value of the output value (not to be used within the module)
op getOutputValue (__, __) : Qid SORT_VHDL_LINE -> Int
  eq getOutputValue (VarName , Lines signal-value (fsmState Length VarValue) ) = VarValue .

--- wrap a value to predetermined length
op wrap (__, __) : Int Int -> Int .
  ceq wrap (VarValue , Length) = wrap (VarValue , (2 ^ (Length - 1)) ) if Length > 0 and VarValue > 2 ^ (Length - 1) .
  ceq wrap (VarValue , Length) = wrap (VarValue + (2 ^ (Length - 1)) ) if Length > 0 and VarValue <= (2 ^ (Length - 1)) .
  --- underflow
  ceq wrap (VarValue , Length) = VarValue [ otherwise ] .
  --- in range

--- set a signal, returns lines with signal set
op setValue (__, __, __) : SORT_VHDL_LINE Qid Int -> SORT_VHDL_LINE .
op setValue (signal-value (VarName Length OldInt) Lines , VarName , VarValue)
  <= signal-value (VarName Length wrap (VarValue , Length) ) Lines
  eq output-value ( output-value (VarName Length OldInt ) Lines , VarName , VarValue) =
  output-value (VarName Length wrap (VarValue , Length ) Lines) .

--- initialize an input value, do not use during execution
op initializeInputValue( VarName Length OldInt ) : SORT_VHDL_LINE Qid Int -> SORT_VHDL_LINE
  
  eq initializeInputValue( input-value(VarName Length OldInt ) Lines , VarName
  , VarValue ) = input-value(VarName Length wrap(VarValue , Length )) Lines .

--- increment cycle count
op incrementCycleCount : SORT_VHDL_LINE -> SORT_VHDL_LINE .
  
  eq incrementCycleCount( Lines cyclecount(VarValue) ) = Lines
  cyclecount(VarValue + 1) .

op iterate() : SORT_VHDL_LINE -> SORT_VHDL_LINE .

--- execution, use getValue to get the value of outputs
op run() : SORT_VHDL_LINE -> SORT_VHDL_LINE .
  
  eq run( Lines ) = iterate(incrementCycleCount(setValue( Lines , 'fsmState , 1))) .

--- execution with input values
op initialValue( VarName Length OldInt ) : Qid Int -> SORT_INITIAL.
  
  eq initialValue( VarName Length OldInt ) = Qid Int .

op run( VarName IntValue ) : SORT_INITIAL -> SORT_VHDL_LINE .
  
  eq run( VarName IntValue ) = run( VarName , IntValue ) .

--- get the cycle count
op getCycleCount() : SORT_VHDL_LINE -> Int.
  
  eq getCycleCount( Count ) = Count .

--- we reverted back to state 0 (should be done, stop iterating)
  
  eq iterate( Lines signal-value('fsmState Length StateNum ) ) = Lines
  signal-value('fsmState Length StateNum ) .

--- Assign Int
op ( state _ assignInt _ ) : Int Qid Int Int -> SORT_VHDL_STATE .
  
  op assignInt( StateNum assignInt VarName IntValue NextStateNum ) =
  assignInt( StateNum VarName IntValue IntValue NextStateNum ) .
  
  eq iterate( Lines assignInt( StateNum VarName IntValue NextStateNum )
  signal-value('fsmState Length StateNum ) )

  = iterate( incrementCycleCount( setValue( Lines
  assignInt( StateNum VarName IntValue NextStateNum )
  signal-value('fsmState Length StateNum )
  , VarName
  , IntValue)
  , 'fsmState
  , NextStateNum ) ) .

--- Assign Signal
op ( state _ assignSignal _ ) : Int Qid Int Int -> SORT_VHDL_STATE .
  
  op assignSignal( StateNum assignSignal VarName OtherVarName NextStateNum ) =
  assignSignal( StateNum VarName OtherVarName NextStateNum ) .
  
  eq iterate( Lines assignSignal( StateNum VarName OtherVarName NextStateNum )
  signal-value('fsmState Length StateNum ) )

  = iterate( incrementCycleCount( setValue( Lines
  assignSignal( StateNum VarName OtherVarName NextStateNum )
  signal-value('fsmState Length StateNum )
  , VarName
  , getInternalValue(OtherVarName, Lines))
  , 'fsmState
  , NextStateNum ) ) .

--- Assign Add Signal Int
op ( state _ assignAddSignalInt _ ) : Int Qid Int Int ->
  SORT_VHDL_STATE .
  
  op assignAddSignalInt( StateNum assignAddSignalInt VarName OtherVarName IntValue
  NextStateNum ) = assignAddSignalInt( StateNum VarName OtherVarName IntValue
  NextStateNum ) .
  
  eq iterate( Lines assignAddSignalInt( StateNum VarName OtherVarName IntValue
  NextStateNum ) signal-value('fsmState Length StateNum ) ) =
iterate (incrementCycleCount(
    setValue(
        setValue(
            Lines assignAddSignal (StateNum VarName OtherVarName IntValue
            , VarName
            , IntValue + getInternalValue(OtherVarName, Lines))
            , 'fsmState
            , nextStateNum)
        ) ) .
-- Assign Add Signal Int
op (state assignAddSignal (.. .. .. ..) : Int Qid Qid Int Int ->
    sortVHDL_STATE
op assignAddSignal (.. .. .. ..) : Int Qid Qid Int Int -> sortVHDL_STATE .
eq (state StateNum assignAddSignal VarName OtherVarName IntValue
    nextStateNum) = assignAddSignal (StateNum VarName OtherVarName
    ThirdVarName nextStateNum) .
eq iterate (Lines assignAddSignal (StateNum VarName OtherVarName
    ThirdVarName nextStateNum) signal-value (‘fsmState Length StateNum) )

-- Assign Mult Signal Int
op (state assignMultSignal (.. .. .. ..) : Int Qid Qid Int Int ->
    sortVHDL_STATE
op assignMultSignal (.. .. .. ..) : Int Qid Qid Int Int -> sortVHDL_STATE .
eq (state StateNum assignMultSignal VarName OtherVarName IntValue
    nextStateNum) = assignMultSignal (StateNum VarName OtherVarName
    IntValue nextStateNum) .
eq iterate (Lines assignMultSignal (StateNum VarName OtherVarName
    IntValue nextStateNum) signal-value (‘fsmState Length StateNum) )

-- Assign Shift Right Signal Int
op (state assignShiftRightSignal (.. .. .. ..) : Int Qid Qid Int Int ->
    sortVHDL_STATE
op assignShiftRightSignal (.. .. .. ..) : Int Qid Qid Int Int -> sortVHDL_STATE .
eq (state StateNum assignShiftRightSignal VarName OtherVarName IntValue
    nextStateNum) = assignShiftRightSignal (StateNum VarName OtherVarName
    IntValue nextStateNum) .
eq iterate (Lines assignShiftRightSignal (StateNum VarName OtherVarName
    IntValue nextStateNum) signal-value (‘fsmState Length StateNum) )

-- Conditional Greater Signal Int
op (state conditionalGreaterSignal (.. .. .. ..) : Int Qid Int Int Int ->
    sortVHDL_STATE
op conditionalGreaterSignal (.. .. .. ..) : Int Qid Int Int Int -> sortVHDL_STATE .
eq (state StateNum conditionalGreaterSignal VarName IntValue
TrueNextStateNum FalseNextStateNum = conditionalGreaterSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum).

cq iterate (Lines conditionalGreaterSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalGreaterSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , TrueNextStateNum))
  ) if getInternalValue (VarName, Lines) > IntValue.

eq iterate (Lines conditionalGreaterSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalGreaterSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , FalseNextStateNum))
  ) [otherwise].

--- Conditional Equals Signal Int
op (state conditionalEqualsSignalInt ) : Int Qid Int Int Int ->
SORT_VHDL_STATE
op conditionalEqualsSignalInt ( ) : Int Qid Int Int Int ->
SORT_VHDL_STATE
eq (state StateNum conditionalEqualsSignalInt VarName IntValue TrueNextStateNum FalseNextStateNum) = conditionalEqualsSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum).

cq iterate (Lines conditionalEqualsSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalEqualsSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , TrueNextStateNum))
  ) if getInternalValue (VarName, Lines) == IntValue.

eq iterate (Lines conditionalEqualsSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalEqualsSignalInt (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , FalseNextStateNum))
  ) [otherwise].

--- Conditional Greater Signal Signal
op (state conditionalGreaterSignalSignal ) : Int Qid Int Int Int ->
SORT_VHDL_STATE
op conditionalGreaterSignalSignal ( ) : Int Qid Int Int Int ->
SORT_VHDL_STATE
eq (state StateNum conditionalGreaterSignalSignal VarName OtherVarName TrueNextStateNum FalseNextStateNum) = conditionalGreaterSignalSignal (StateNum VarName OtherVarName TrueNextStateNum FalseNextStateNum).

cq iterate (Lines conditionalGreaterSignalSignal (StateNum VarName OtherVarName TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalGreaterSignalSignal (StateNum VarName OtherVarName TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , TrueNextStateNum))
  ) if getInternalValue (VarName, Lines) > getInternalValue (OtherVarName, Lines).

eq iterate (Lines conditionalGreaterSignalSignal (StateNum VarName OtherVarName TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)) =
  iterate (incrementCycleCount(
    setValue (Lines conditionalGreaterSignalSignal (StateNum VarName OtherVarName TrueNextStateNum FalseNextStateNum) signal = value ('fsmState Length StateNum)
      , 'fsmState
      , FalseNextStateNum))
  ) [otherwise].
A.5 Roll VMaude Code

The VMaude code for the rollover decision module is listed here.

```plaintext
A.5 Roll VMaude Code

The VMaude code for the rollover decision module is listed here.
```

--- Conditional Lesser Signal Int
  op (state | conditionalLesserSignalInt | ...): Int Qid Int Int Int
  op conditionalLesserSignalInt (...): Int Qid Int Int Int

  eq (state StateNum conditionalLesserSignalInt VarName IntValue
      TrueNextStateNum FalseNextStateNum) = conditionalLesserSignalInt (StateNum
      VarName IntValue TrueNextStateNum FalseNextStateNum).

  eq iterate (Lines conditionalLesserSignalInt (StateNum VarName IntValue
      TrueNextStateNum FalseNextStateNum) signal-value ('fsmState Length StateNum)
      = iterate (incrementCycleCount (setValue (Lines conditionalLesserSignalInt
          (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum)
          signal-value ('fsmState Length StateNum), 'fsmState
          TrueNextStateNum))
      ) if getInternalValue (VarName, Lines) < IntValue.

  eq iterate (Lines conditionalLesserSignalInt (StateNum VarName IntValue
      TrueNextStateNum FalseNextStateNum) signal-value ('fsmState Length StateNum)
      = iterate (incrementCycleCount (setValue (Lines conditionalLesserSignalInt
          (StateNum VarName IntValue TrueNextStateNum FalseNextStateNum)
          signal-value ('fsmState Length StateNum), 'fsmState
          FalseNextStateNum)) [otherwise].
endm

A.5 Roll VMaude Code

The VMaude code for the rollover decision module is listed here.

```plaintext
A.5 Roll VMaude Code

The VMaude code for the rollover decision module is listed here.
```
A.6 Roll VHDL Code

The generated VHDL code for the rollover module is listed here.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

candidate entity input is
  port (
    clock : in std_logic := '0';
    start : in std_logic := '0';
    done : out std_logic := '0';
    iSlope : in signed(15 downto 0) := (others => '0');
    iNextSteering : in signed(15 downto 0) := (others => '0');
    iNextVelocity : in signed(15 downto 0) := (others => '0');
    allow : out signed(0 downto 0) := (others => '0');
  );

architecture Behavioral of input is
  signal slope : signed(15 downto 0) := (others => '0');
  signal nextSteering : signed(15 downto 0) := (others => '0');
  signal fsmState : integer range 0 to 255 := 0;
  signal worstCaseSlope : signed(15 downto 0) := (others => '0');
  signal rightHandSide : signed(15 downto 0) := (others => '0');
  signal leftHandSide : signed(31 downto 0) := (others => '0');
begin
  process (clock) begin
    if rising_edge (clock) then
      case fsmState is
        when 0 =>
          done <= '0';
        when others =>
          fsmState <= 1;
      end case;
    end if;
    when 1 =>
      if newvel <= 5000 then
        check line equation
        (state 20 assignSlope 'leftHandSide' 'nextSteering 21')
        (state 21 assignSlope 'rightHandSide' 'nextVelocity 19 20')
        if NewVel > 5000 check newsteer in 19
        (state 19 conditionalLesserSignalInt 'nextSteering 700 19 20')
        if Steering < 700 'allow', else deny
        newvel <= 5000, check line equation
        (state 22 assignSlope 'rightHandSide' 'rightHandSide 146 23')
        if S state 8 ', worstCaseSlope is 800
        (state 8 conditionalGreaterSignalInt 'nextVelocity 5000 19 20')
        if NewVel > 5000 check newsteer in 19
        (state 9 conditionalGreaterSignalInt 'nextVelocity 2900 27 10')
        if newvel <= 2900 deny
      end if;
    end process;
end Behavioral;
```

A.6 Roll VHDL Code

The generated VHDL code for the rollover module is listed here.
if iNextSteering > 0 then
    fsmState <= 52;
else
    fsmState <= 54;
end if;
when 52 =>
    slope <= to_signed(to_integer(iSlope), 16);
    fsmState <= 53;
when 53 =>
    nextSteering <= to_signed(to_integer(iNextSteering), 16);
    fsmState <= 41;
when 54 =>
    slope <= to_signed(to_integer(slope) * (-1), 16);
    fsmState <= 55;
when 55 =>
    nextSteering <= to_signed(to_integer(nextSteering) * (-1), 16);
    fsmState <= 56;
when 41 =>
    worstCaseSlope <= slope + (200);
    fsmState <= 42;
when 42 =>
    if worstCaseSlope > 1000 then
        fsmState <= 43;
    else
        fsmState <= 44;
    end if;
when 43 =>
    worstCaseSlope <= to_signed(1000, 16);
    fsmState <= 44;
when 44 =>
    if worstCaseSlope = 600 then
        fsmState <= 7;
    else
        fsmState <= 45;
    end if;
when 45 =>
    if worstCaseSlope = 800 then
        fsmState <= 8;
    else
        fsmState <= 46;
    end if;
when 46 =>
    if worstCaseSlope = 1000 then
        fsmState <= 9;
    else
        fsmState <= 10;
    end if;
when 7 =>
    rightHandSide <= to_signed(to_integer(nextVelocity), 32);
    fsmState <= 12;
when 12 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (51), 32);
    fsmState <= 13;
when 13 =>
    rightHandSide <= rightHandSide srl (8);
    fsmState <= 14;
when 14 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (-1), 32);
    fsmState <= 15;
when 15 =>
    rightHandSide <= rightHandSide + (2400);
    fsmState <= 16;
when 16 =>
    leftHandSide <= to_signed(to_integer(nextSteering), 32);
    fsmState <= 17;
when 17 =>
    if leftHandSide > rightHandSide then
        done <= '1';
    else
        fsmState <= 10;
    end if;
when 8 =>
    if nextVelocity > 5000 then
        fsmState <= 19;
    else
        fsmState <= 20;
    end if;
when 19 =>
if nextSteering < 700 then
    fsmState <= 10;
else
    done <= '1';
    fsmState <= 0;
end if;
when 20 =>
    leftHandSide <= to_signed(to_integer(nextSteering), 32);
    fsmState <= 21;
when 21 =>
    rightHandSide <= to_signed(to_integer(nextVelocity), 32);
    fsmState <= 22;
when 22 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (146), 32);
    fsmState <= 23;
when 23 =>
    rightHandSide <= rightHandSide srl (8);
    fsmState <= 24;
when 24 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (-1), 32);
    fsmState <= 25;
when 25 =>
    rightHandSide <= rightHandSide + (3557);
    fsmState <= 26;
when 26 =>
    if leftHandSide > rightHandSide then
        done <= '1';
        fsmState <= 0;
    else
        fsmState <= 10;
    end if;
when 9 =>
    if nextVelocity > 2900 then
        fsmState <= 27;
    else
        fsmState <= 10;
    end if;
when 27 =>
    if nextSteering > 300 then
        fsmState <= 28;
    else
        fsmState <= 10;
    end if;
when 28 =>
    leftHandSide <= to_signed(to_integer(nextSteering), 32);
    fsmState <= 29;
when 29 =>
    rightHandSide <= to_signed(to_integer(nextVelocity), 32);
    fsmState <= 30;
when 30 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (77), 32);
    fsmState <= 31;
when 31 =>
    rightHandSide <= rightHandSide srl (8);
    fsmState <= 32;
when 32 =>
    rightHandSide <= to_signed(to_integer(rightHandSide) * (-1), 32);
    fsmState <= 33;
when 33 =>
    rightHandSide <= rightHandSide + (1770);
    fsmState <= 34;
when 34 =>
    if leftHandSide > rightHandSide then
        done <= '1';
        fsmState <= 0;
    else
        fsmState <= 10;
    end if;
when 10 =>
    allow <= to_signed(-1, 1);
    done <= '1';
    fsmState <= 0;
when others =>
end case;
end if;
end process;
end Behavioral;
A.7 VHDL Generator Code

The VMaude to VHDL generator, contains several files, mostly dealing with parsing and generating VHDL. The source file with the main logic for doing the translation is provided below.

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.TreeMap;

/**
 * Stan Bak
 * 6/16/2009
 * VHDL-structured Maude to VHDL Translator
 * Sample of VHDL-Structured Maude:
 * (cycle-count) (input 'slope 16) (input 'slope2 16) (output 'output 8)
 * (signal 'fsmState 8) (signal 'internal 8) (signal 'internal2 4)
 * (state 1 assignInt 'internal 9 2) (state 2 assignInt 'internal 9 3)
 * (state 3 conditionalGreaterSignal 'slope 'slope2 4 5) (state 4 assignInt 'output 4 0)
 * (state 5 assignInt 'output 5 0)
 */
public class Main {
    private static VHDL.Entity entity = null;
    private static TreeMap<String, Integer> signalToSizeMap = new TreeMap<String, Integer>();
    public static void main(String[] args) {
        String inputFilename = "input.txt";
        String outputFilename;
        String baseName = "SafeBehavior";
        String behavioralFilename = "beh.txt";
        if (args.length > 0)
            inputFilename = args[0];
        int periodIndex = inputFilename.indexOf('.');
        if (periodIndex != -1)
            baseName = inputFilename.substring(0, periodIndex);
        outputFilename = baseName + ".vhd";
        entity = new VHDL.Entity(baseName);
        entity.createBlank(baseName);
        try {
            BufferedReader fin = new BufferedReader(new InputStreamReader(new FileInputStream(inputFilename)));
            BufferedWriter fout = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(behavioralFilename)));
            initialWork(fout);
            for (String line = fin.readLine(); line != null; line = fin.readLine()) {
                // trim comments
                int commentIndex = line.indexOf("--");
                if (commentIndex != -1)
                    line = line.substring(0, commentIndex);
                fout.write(line + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "---" + "
" + "---
" + "."
```

72
```java
// trim whitespace
line = line.trim();
if (line.length() == 0) // blank line
    continue;

// check for parentheses
if (line.length() < 2 || line.charAt(0) != '(' || line.charAt(line.length() - 1) != ')')
{
    System.err.println("Error: line does not both start and end with parentheses: "+ line + "+");
    continue;
}

// trim parentheses
int len = line.length();
line = line.substring(1, len - 1);

// finally print out the result
processLine(line, fout);
}
finalWork(fout);
fin.close();
fout.close();

// set entity behavioral file
entity.addBehavior(behavioralFilename);

// okay now print the entity
entity.printEntity(outputFilename);
System.out.println("Done!");
catch (Exception e)
{
    System.err.println(e);
}
System.exit(0);

/**
 * Set up entity and behavioral file
 * @param fout the behavior file buffered writer to print the vhdl to
 * @throws Exception if there's a file I/O error
 */
private static void initialWork(BufferedWriter fout) throws Exception
{
    entity.addSignal(new VHDL.Signal("clock", "std_logic", SignalDirection.IN, "'0'"));
    fout.write("process (clock) begin
    if rising_edge(clock) then
      case fsmState is
        when 0 =>
          done <= '0';
        when others =>
          end case;
      end if;
    end process;

    entity.addSignal(new VHDL.Signal("start", "std_logic", SignalDirection.IN, "'0'"));
    setSignalSize("start", 1);
    entity.addSignal(new VHDL.Signal("done", "std_logic", SignalDirection.OUT, "'0'"));
    setSignalSize("done", 1);
}

/**
 * Complete the Behavioral file
 * @param fout the behavior file buffered writer to print the vhdl to
 * @throws Exception if there's a file I/O error
 */
private static void finalWork(BufferedWriter fout) throws Exception
{
    fout.write("end case;\nend if;\nend process;\n")
```
private static void fsmTransition(int tabs, String to, BufferedWriter fout) throws Exception
{
    int state = Integer.parseInt(to);
    if (state == 0) // done!
    {
        for (int x = 0; x < tabs; ++x)
            fout.write("\t");
        fout.write(" done\n");
    }
    for (int x = 0; x < tabs; ++x)
        fout.write("\t");
    fout.write(" fsmState <= "+ state + ";\n");
}

private static void processLine(String line, BufferedWriter fout) throws Exception
{
    String tokens[] = line.split(" ");
    try
    {
        if (tokens.length == 1 && tokens[0].equals("cycle\-count")) // do nothing
        
        else if (tokens.length == 3 && tokens[0].equals("input") && tokens[1].length() > 1)
        {
            // (input \'slope 16)
            String name = tokens[1].substring(1);
            int len = Integer.parseInt(tokens[2]);
            setSignalSize(tokens[1], len);
            entity.addSignal(new VHDL.Signal(name,
                  "signed(" + (len-1) + "+ downto 0)", SignalDirection.IN, "(others => '0')") );

        }
        else if (tokens.length == 3 && tokens[0].equals("output") && tokens[1].length() > 1)
        {
            // (output \'output 8)
            String name = tokens[1].substring(1);
            int len = Integer.parseInt(tokens[2]);
            setSignalSize(tokens[1], len);
            entity.addSignal(new VHDL.Signal(name,
                  "signed(" + (len-1) + "+ downto 0)", SignalDirection.OUT, "(others => '0')") );

        }
        else if (tokens.length == 3 && tokens[0].equals("signal") && tokens[1].equals("\'fsmState\")
        {
            // (signal \'appple 8)
            String name = tokens[1].substring(1);
            int len = Integer.parseInt(tokens[2]);
            setSignalSize(tokens[1], len);
            int rangeMax = (int)Math.round(Math.pow(2, len)) - 1;
            entity.addSignal(new VHDL.Signal("fsmState", "integer range 0 to "+ rangeMax, SignalDirection.INTERNAL, "0") );

        }
        else if (tokens.length >= 3)
        {
            // (signal \'appple 8)
            String name = tokens[1].substring(1);
            int len = Integer.parseInt(tokens[2]);
            setSignalSize(tokens[1], len);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
entity addSignal \( \text{new VHDLSignal(name,} \\
\text{"signed(" + (len-1) + " down to 0")}, \text{SignalDirection.INTERNAL, } \text{"(others => '0')} \text{")}) ;} \\
\text{}} \\
else if (tokens.length > 4 \&\& tokens[0].equals("state")) \\
\{ \\
\text{int stateNum = Integer.parseInt(tokens[1]) ;} \\
\text{fout.write("\"token " + stateNum + " =>\n\"t" ;)} \\
\text{if (tokens[2].equals("assignInt"))} \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{fout.write(name + " <= to\nsigned(" +} \\
tokens[4] + ", " + getSignalSize(tokens[3]) + ");\n") ;} \\
\text{fsmTransition(2, tokens[5], fout) ;} \\
\text{else if (tokens[2].equals("assignSignal"))} \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{String rhs = tokens[4].substring(1) ;} \\
\text{fout.write(name + " <= " + rhs + ");\n") ;} \\
\text{fsmTransition(2, tokens[5], fout) ;} \\
\} \\
else if (tokens[2].equals("assignAddSignalInt")) // add is okay to simulate without casting \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{String rhsSig = tokens[4].substring(1) ;} \\
\text{fout.write(name + " <= " + rhsSig + ");\n") ;} \\
\text{fsmTransition(2, tokens[6], fout) ;} \\
\} \\
else if (tokens[2].equals("assignAddSignalSignal")) // add is okay to simulate without casting \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{String rhsSig = tokens[4].substring(1) ;} \\
\text{String rhsSig2 = tokens[5].substring(1) ;} \\
\text{fout.write(name + " <= " + rhsSig + rhsSig2 + ");\n") ;} \\
\text{fsmTransition(2, tokens[6], fout) ;} \\
\} \\
else if (tokens[2].equals("assignMultSignalInt")) // mult needs casting for correct simulation \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{String rhsSig = tokens[4].substring(1) ;} \\
\text{String rhsSig2 = tokens[5].substring(1) ;} \\
\text{fout.write(name + " <= " + rhsSig + getSignalSize(tokens[4]) + ");\n") ;} \\
\text{fsmTransition(2, tokens[6], fout) ;} \\
\} \\
else if (tokens[2].equals("assignShiftRightSignalInt")) // srl simulates okay without cast \\
\{ \\
\text{String name = tokens[3].substring(1) ;} \\
\text{String rhsSig = tokens[4].substring(1) ;} \\
\text{fout.write(name + " <= " + rhsSig + ");\n") ;} \\
\text{srl(" + tokens[5] + ");\n") ;} \\
\text{fsmTransition(2, tokens[6], fout) ;} \\
\} \\
else if (tokens[2].equals("conditionalGreaterSignalInt")) \\
\{ \\
\text{String sig = tokens[3].substring(1) ;} \\
\text{fout.write("if " + sig + " > " + tokens[4] + ");\n") ;} \\
\text{fsmTransition(3, tokens[5], fout) ;} \\
\text{fout.write("\"t\"tend if;\n") ;} \\
\} \\
else if (tokens[2].equals("conditionalEqualsSignalInt")) \\
\{ \\
\text{String sig = tokens[3].substring(1) ;} \\
\text{fout.write("if " + sig + " == " +} 

tokens[4] + " then\n")
        fout.write("if "+sig+" > "+
        fsmTransition(3,tokens[5],fout);
        fout.write("\t\tend if;\n")
    } else if 
    {tokens[2].equals("conditionalGreaterSignalSignal")
        String sig = tokens[3].substring(1);
        String otherSig =
tokens[4].substring(1);
        fout.write("if "+sig+" > "+
        fsmTransition(3,tokens[5],fout);
        fout.write("\t\telse
        fsmTransition(3,tokens[6],fout);
        fout.write("\t\tend if;\n")
    } else if 
    {tokens[2].equals("conditionalLesserSignalInt")
        String sig = tokens[3].substring(1);
        fout.write("if "+sig+" < "+
        tokens[4] + " then\n")
        fout.write("if "+sig+" < "+
        fsmTransition(3,tokens[5],fout);
        fout.write("\t\telse
        fsmTransition(3,tokens[6],fout);
        fout.write("\t\tend if;\n")
    } else 
    {throw new Exception("(2) Unprocessable line: "+line + "\n")
    }
    } else 
    {throw new Exception("(1) Unprocessable line: "+
        + line + "\n")
    }
    }
    catch (Exception e)
    {
        System.out.println("Parse Error on line "+line + "\n")
        throw e;
    }
    } /*
    * Set the size for a signal, which can then be retrieved using
    * getSignalValue
    * @param signalName the name of the signal, with the Qid
    * @param size the size of the associated signal
    * @throws Exception if the signal already has a size set
    */
    private static void setSignalSize(String signalName, int size) throws
    Exception
    {
        if (signalToSizeMap.get(signalName) != null)
            throw new Exception("Signal "+signalName+" already has its size set.");
        signalToSizeMap.put(signalName, size);
    }
    } /*
    * Get the size of a signal, previously set with setSignalSize
    * @param signalName the name of the signal we want the size of, with the Qid
    * @return the size of the signal
    * @throws Exception if the signal's size was not previously set with
    * setSignalSize
    */
    private static int getSignalSize(String signalName) throws Exception
    {
        Integer i = signalToSizeMap.get(signalName);
        if (i == null)
            throw new Exception("Signal "+signalName+" does not have a set size.");
    }
return i.intValue();
References


