Hybrid Cyberphysical System Verification With Simplex Using Discrete Abstractions

Stanley Bak[ψ], Ashley Greer[⊕], Sayan Mitra[ψ]
University of Illinois at Urbana-Champaign[ψ], Deere & Company[⊕]
sbak2@illinois.edu, GreerAshleyE@johndeere.com, mitras@crhc.uiuc.edu

***Abstract*—Providing integrity, efficiency, and performance guarantees is a key challenge in the development of next-generation cyberphysical systems. Rather than mandating complete system verification, the Simplex Architecture provides robust designs by incorporating a supervisory controller that takes corrective action only when the system is in danger of violating a desired invariant property such as safety. The central issue in applying this approach is designing the switching logic for the supervisory controller such that it guarantees safety and at the same time is not overly conservative. Previous research in the area relied on finding Lyapunov functions for the underlying continuous dynamical system. In contrast, in this paper, we present an automatic method for solving this design problem through discrete abstractions of the underlying hybrid system and model checking. We present a case study where, in collaboration with John Deere, we use the developed approach to create the Simplex decision module for an off-road vehicle, which is formally verified as both correct and timely.**

I. Introduction

In military and aviation applications, robustness and reliability are achieved through sufficient redundancy and rigorous development and certification processes such as the DO-178B [1]. Such means and methodologies, however, are untenable for several other industries including automotive, agriculture, and off-road vehicles, because of a product cost difference of three orders of magnitude. Consider, for instance, the agricultural vehicle industry. A modern tractor has sophisticated computer control algorithms implemented on upwards of 25 microprocessor-based controllers, providing features ranging from automatic climate control to automatic guidance. As the labor market shrinks, it is expected that vehicles of the future will provide more automation and high-level coordination capabilities. Automation, however, must be reliable and ideally verified, but the high cost of complete system verification can be prohibitive.

In order to address the issue of verification, while simultaneously taking into account cost and practicality, we promote the use of the Simplex Architecture [2]. The Simplex Architecture uses *simplicity to control complexity*, where a simple safety controller is combined with the existing high-level complex controller. Under normal operation, the decision module uses the complex controller, and only switches to the conservative safety controller when monitored properties are in danger of becoming violated. The safety controller will then drive the system to a safer state, where the complex controller can again regain control. The key challenge of such an architecture, however, is correctly developing the decision module to switch controllers so that the properties of concern can never become violated[1]. By leveraging on hybrid systems modeling and analysis techniques, we confront this main challenge of Simplex in this paper.

Previous Simplex work has proposed two approaches for correct system construction, a control-theoretic approach for verification of continuous systems [2], and a best-effort development strategy where traditional software engineering techniques and thorough testing are used to create the decision module's switching logic [3]. The first can not be applied for systems with discrete state, or systems with mixed discrete and continuous dynamics (hybrid systems). The second can not be used to formally prove properties about the resultant systems. In this paper, we present a technique which is applicable to a larger class of systems including a restricted class of hybrid systems, while still permitting formal system verification. We show that the technique is useful through a case study we performed with John Deere regarding autonomous off-road vehicle verification.

The main contributions of this paper are:

- An algorithmic construction of abstract finite discrete transition systems from a restricted class of hybrid systems which simulate the original hybrid system, and can therefore be used for property verification;
- The use of the constructed discrete transition system to determine the correct switching point for the Simplex decision module, as well as the verification of the combined system;
- A case study which applies the verification technique to prevent rollover for an autonomous off-road vehicle system. The output of the case study is verified correct and timely VHDL implementation of the Simplex decision module.

This paper is organized as follows. First, in Section II, we review the Simplex architecture, and discuss previous verification methods. In Section III, we outline hybrid systems, discrete transition systems, and formally specify Simplex as a hybrid automaton. Then, we discuss our hybrid system restrictions, and the construction of the corresponding discrete

[1]Another progress guarantee is also necessary, where we have to guarantee that the complex controller will always eventually be allowed to regain control of the system. We do not focus on formally proving such progress properties in this paper.

transition system in Section IV. Next, in Section V, we present an off-road vehicle case study which demonstrates using a model checker and an automatically constructed abstract discrete transition system to verifiably create the Simplex decision module. We finish with related research in Section VI, and conclusions in Section VII.

II. The Simplex Architecture

The Simplex Architecture incorporates supervisory control and switching logic on top of an unverified controller, for the purpose of improving system robustness. We first present an overview of the architecture's main components in Section II-A, and then discuss correctness and verification in Section II-B.

A. Simplex Overview

A Simplex system consists of three main components. Under normal operating conditions, the unverified *complex controller* actuates the system. If the system state becomes in danger of property violation, the *safety controller* takes over. After some time, the safety controller should drive the system to a state that can tolerate aggressive action without danger of property violation, and the complex controller is allowed to resume control. The switching between the controllers is performed by a *decision module*.

The Simplex architecture has previously been applied to improve the safety of a fleet of remote-controlled cars [4], a pacemaker[5], and a set of advanced aircraft maneuvers [6]. The advantage of such a design is in the potential for simpler verification. By verifying the safety controller and decision module, properties about the composite system can be proven. Thus, we avoid having to perform verification on the complex controller directly, which, in some instances, can significantly reduce verification effort. Creating the safety controller is usually less intensive than creating the complex controller, as only safety-critical considerations need to be taken into account, whereas the complex controller must also meet mission-critical requirements.

Simplex, however, should not be regarded as a one-size-fits-all robustness solution, as it only addresses logical program errors[2], and, it is only applicable to some systems. Particularly, the system must be adequately modeled, properties of interest must be defined, a simpler-to-verify safety controller must be created, and the associated decision module must be verified. Here, we focus on the last of these requirements, constructing a verified decision module. That is, we assume we are provided with a system model, specific properties to verify, and a safety controller. We now discuss existing approaches for producing a correct Simplex decision module.

B. Decision Module Correctness

From the design of Simplex, it is clear that a correct decision module is essential. Some systems can not modeled easily with differential equations or have safety controllers that are too complex to permit proofs for useful properties [3]. In these instances, best practices from software engineering may be used to develop the decision module. In this paper, however, we aim to provide formal guarantees.

[2]For hardware faults and transient errors, compatible techniques such as triple modular redundancy (TMR) [7] can be used.

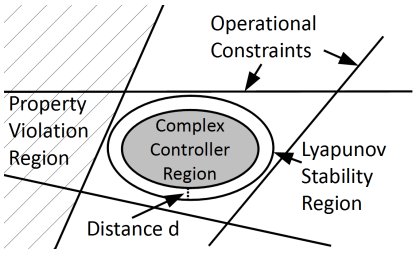

Fig. 1. In a continuous two-dimensional state space, a Lyapunov function of the safety controller can be used to guarantee safety. The distance d is at least the maximum gradient of the state space within the stability region times the control iteration time. If the state is within the grey region, the complex controller can be used. Otherwise, the safety controller is used.

For verifying a decision module in a system with continuous dynamics, a Lyapunov function's stability region can be used drive the decision module [2]. A Lyapunov function for a controller for a particular plant defines an n-dimensional ellipsoid within an n-dimensional state space where, if the current system state is within the ellipsoid and the controller is used, the system will converge to a setpoint. If a Lyapunov function exists for the safety controller where the stability region is defined within the operational constraints without containing property violation states, and we know the maximum gradient over time for any controller within the Lyapunov stability region, we can formally derive the decision module switching rule. If our current system state is at least the maximum gradient times the control iteration time away from the Lyapunov stability region boundary, the complex controller can be used. In the worst case, the state space will proceed at the maximum gradient towards the Lyapunov stability region boundary, but will not cross it before the next control iteration. We can then switch to the safety controller which is guaranteed to converge to the setpoint without leaving the Lyapunov stability region. In this way, the property violation region is never entered. The situation is shown in a two-dimensional state space in Figure 1. In practice, a margin larger than the maximum gradient times the control iteration time is often used to account for modeling errors and provide additional safety. After the safety controller returns the state to within the complex controller region, the complex controller can resume driving the plant. Typically, however, a smaller region is used when switching back to the complex controller, creating a hysteresis which prevents frequent controller switching. The drawback of this approach is that, even within strictly continuous systems, there is no general technique to determine a Lyapunov function for a particular controller and plant.

Another approach for verifiable correctness was taken in our own previous work [5], where Simplex was applied to a purely discrete system, a pacemaker model. Here, the entire plant model and safety controller were expressed with a discrete transition system and input into a model checker. The model

checker made sure that, for every possible system state and every possible complex controller command, the future state of the system did not violate the pacemaker-specific safety properties such as beating too slowly or changing heart rate too rapidly. This approach only works if the system model is completely discrete and has no continuous elements, and is tractable by a model checker.

In the remaining sections, we outline and evaluate a new technique for verifiable Simplex correctness. In particular, our method targets hybrid systems which may contain both discrete and continuous dynamics.

III. Hybrid Systems Preliminaries

Let V be a set of variables. Each variable $v \in V$ is associated with a *type* which defines the set of values v can take. The set of valuations of V is denoted by $val(V)$. A variable may be *discrete* or *continuous*. Typically, discrete variables model protocol or software state, and continuous variables model physical quantities such as time, position, and velocity. For a subset $S \subseteq \mathbb{R}^n$, we denote by S_i the projection of S on the i^{th} coordinate, $0 \leq i \leq n$.

A. Hybrid Automata and Discrete Abstractions

Discrete abstractions for cyber-physical systems are mathematically modeled as automata or labeled transition systems.

Definition 1. *A* labeled transition system (LTS) $\mathcal{A}$ *is a tuple* $\langle Y, \Theta, A, \mathcal{D} \rangle$ *where*

(a) Y *is a set of variables; elements of* $val(Y)$ *are called* states*,*
(b) $\Theta \subseteq val(Y)$ *is a set of* initial states*,*
(c) A *is a set of* actions*, and*
(d) $\mathcal{D} \subseteq val(Y) \times A \times val(Y)$*, is a set of transitions. For an element* $(\mathbf{y}, a, \mathbf{y}') \in \mathcal{D}$*, we write* $\mathbf{y} \xrightarrow{a} \mathbf{y}'$.

An *execution* of DS $\mathcal{A}$ is an (possibly infinite) alternating sequence $\mathbf{y}_0 a_1 \mathbf{y}_1 a_2 \ldots$, where $\mathbf{y}_0 \in \Theta$, and for each i, $\mathbf{y}_i \xrightarrow{a_{i+1}} \mathbf{y}_{i+1}$. The set of executions and reachable states of $\mathcal{A}$ are denoted by $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$. Given a set of states $S \subseteq val(Y)$, the set $\mathsf{Prev}_{\mathcal{A}}(S)$ is defined as the set of states from which some state in S can be reached in a single transition of $\mathcal{A}$, that is, $\{\mathbf{y} \in val(Y) \mid \exists \mathbf{y}' \in S, a \in A, \mathbf{y} \xrightarrow{a} \mathbf{y}'\}$. $\mathsf{BackReach}_{\mathcal{A}}(S)$ is the least fixpoint of the $\mathsf{Prev}_{\mathcal{A}}()$ starting from S.

In this paper, we work with input/output-free hybrid automaton models for cyber-physical systems. A Hybrid Automaton (HA) is a non-deterministic state machine whose variables may change (a) instantaneously through transitions, or (b) continuously over an interval of time following a *trajectory*. A *trajectory* for a set of variables V models continuous evolution of the values of the variables over an interval of time. Formally, a trajectory τ is a map from a left-closed interval of $\mathbb{R}_{\geq 0}$ with left endpoint 0 to $val(V)$. The *first state* of τ, $\tau.\mathsf{fstate}$, is $\tau(0)$. A trajectory τ is *closed* if the domain of τ is $[0, t]$ for some $t \in \mathbb{R}_{\geq 0}$, and we define the last state in τ, $\tau(t)$, as $\tau.\mathsf{lstate}$.

Definition 2. *A* Hybrid Automaton (HA) $\mathcal{A}$ *is a tuple* $(V, \Theta, A, \mathcal{D}, \mathcal{T})$ *where*

(a) V *is a set of variables; the elements of* $val(V)$ *are called* states*,*
(b) $\Theta \subseteq val(X)$ *is the set of* start states*.*
(c) A *is a set of actions.*
(d) $\mathcal{D} \subseteq val(V) \times A \times val(V)$ *is a set of* transitions*. A transition* $(\mathbf{v}, a, \mathbf{v}') \in \mathcal{D}$ *is written in short as* $\mathbf{v} \xrightarrow{a}_{\mathcal{A}} \mathbf{v}'$ *or as* $\mathbf{v} \xrightarrow{a} \mathbf{v}'$ *when* $\mathcal{A}$ *is clear from the context.*
(e) And $\mathcal{T}$ *is set of trajectories for* V *that is closed under prefix, suffix, and concatenation [8]. In addition,* $\mathcal{A}$ *is non-blocking, that is, at any state* $\mathbf{v} \in val(V)$*, either an action can occur or time can elapse.*

An *execution fragment* of $\mathcal{A}$ is a finite or infinite sequence $\tau_0 a_1 \tau_1 a_2 \ldots$, such that for all i in the sequence, $\tau_i.\mathsf{lstate} \xrightarrow{a_{i+1}} \tau_{i+1}.\mathsf{fstate}$. An execution fragment is an *execution* if $\tau_0.\mathsf{fstate} \in \Theta$. The first state of α, $\alpha.\mathsf{fstate}$, is $\tau_0.\mathsf{fstate}$, and for a closed α, its last state, $\alpha.\mathsf{lstate}$, is the last state of its last trajectory. The set of executions and reachable states of $\mathcal{A}$ are denoted by $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$.

Definition 3. *Given a hybrid automaton* $\mathcal{A} = \langle V, \Theta, A, \mathcal{D}, \mathcal{T} \rangle$*, a set of variables* Y*, and a mapping* $f : val(V) \to val(Y)$*, a* discrete abstraction *of* $\mathcal{A}$ *is a labeled transition system* $\mathcal{B} = \langle Y, \Theta', A \cup \{\mathsf{Time}\}, \mathcal{D}' \rangle$*, where*

(a) $\mathbf{y} \in \Theta'$ *iff* $\exists \mathbf{v} \in \Theta, f(\mathbf{v}) = \mathbf{y}$*,*
(b) $(\mathbf{y}, a, \mathbf{y}') \in D'$ *iff*
  (i) transition $a \in A$ *and* $\exists (\mathbf{v}, a, \mathbf{v}') \in \mathcal{D}$ *such that* $f(\mathbf{v}) = \mathbf{y}$ *and* $f(\mathbf{v}') = \mathbf{y}'$*, or*
  (ii) trajectory $a = \mathsf{Time}$ *and* $\exists \tau \in \mathcal{T}$*, such that* τ *is closed and* $f(\tau.fstate) = \mathbf{y}$ *and* $f(\tau.lstate) = \mathbf{y}'$.

In other words, $\mathcal{B}$ is a time abstract transition system that simulates $\mathcal{A}$: every discrete transition $\mathbf{v} \xrightarrow{a}_{\mathcal{A}} \mathbf{v}'$ of $\mathcal{A}$ is simulated by a discrete transition $f(\mathbf{v}) \xrightarrow{a}_{\mathcal{B}} f(\mathbf{v}')$ of $\mathcal{B}$, and every closed trajectory τ of $\mathcal{A}$ is simulated by $f(\tau.\mathsf{fstate}) \xrightarrow{\mathsf{Time}}_{\mathcal{B}} f(\tau.\mathsf{lstate})$ of $\mathcal{B}$.

B. Hybrid model of Simplex

In constructing hybrid automata models for Simplex-based embedded control systems, we consider automata in which an opportunity for high-level control mode switches arise once every Δ unit of time. Such hybrid automata often arise in modeling periodically scheduled real-time and embedded systems [9]. In this section, we define a class of hybrid automaton with a $mode$ variable, which is suitable for capturing Simplex-based control systems. The possible values of $mode$ correspond to the active controller, either the safety controller or the complex controller.

Automaton Simplex, shown in Figure 2, captures an abstract n-dimensional Simplex-based control system. The Simplex specification is parameterized by a positive constant Δ. The three variables are (a) *continuous state* variables $X = \{x_1, \ldots, x_n\}$ each of type $\mathbb{R}$, (b) discrete state variable $mode$ which can take two values, namely $\{\mathsf{safety}, \mathsf{complex}\}$, initialized to safety, and (c) clock variable c, initialized to 0. We denote the vector of n continuous variables as a single x. We say that the system is in the safety (complex) mode

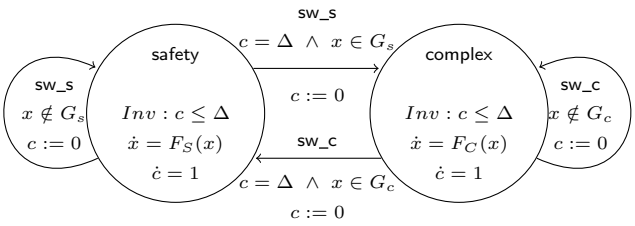

Fig. 2. The Simplex hybrid automaton switches between two Simplex modes.

when $mode$ = safety (and complex, respectively). When the system is in the safety (complex) mode, the continuous variables evolve according to the differential equations $\dot{x} = F_S(x); \dot{c} = 1$ ($\dot{x} = F_C(x); \dot{c} = 1$), where the right hand side of the differential equations F_S (F_C) are real-valued continuous functions, and subject to restrictions that we outline later in Section IV-A. Furthermore, the length of all trajectories of Simplex are upperbounded by Δ.

Every Δ time exactly one action occurs, sw_s or sw_c. As a result, the corresponding discrete transition Simplex may or may not change the value of $mode$; the value of x does not change. The value of $mode$ changes from safety to complex if, at the pre-state of the transition, the valuation of x is in G_c. Similarly, $mode$ changes from complex to safety if x is in G_s in the pre-state.

Thus, given the dynamics of the system (F_s and F_c) under the safe and the complex controllers, and a safety property specified by an unsafe set U, the task of designing the decision module for a Simplex-based control system boils down to finding G_s and G_c such that (a) *(safety)* $\mathsf{Reach}_{\mathsf{Simplex}} \cap U = \emptyset$, and (b) *(utility)* as time goes to infinity, the duration of time for which $mode$ = complex also goes to infinity. As mentioned before, however, in this paper we only focus on formally proving the safety property.

For the simplicity of exposition, we have described Simplex with two modes each of which has continuous dynamics described by differential equations. The design methodology described in the next section, however, generalizes to systems where the safety and the complex mode dynamics are described their own hybrid automaton, with several (sub-)modes with uncontrolled[3] discrete switches.

IV. Quantization-based Discrete Abstractions

In this section, after discussing the restrictions we place on the hybrid automata associated with each mode (Section IV-A), we use a quantization-based abstraction function (Section IV-B) to algorithmically construct discrete abstractions of the individual modes of a Simplex hybrid automata (Section IV-C), which are then used to verifiably determine the $mode$ guards, G_c and G_s of the original Simplex system (Section IV-D).

Recall, the set of variables for Simplex are $V = \{x, mode, c\}$, where x is a continuous variable of type $\mathbb{R}^n$, $mode$ is a $\{\mathsf{safety}, \mathsf{complex}\}$-valued discrete variable, and c is a real-valued clock. We define $\mathsf{Simplex}_{\mathsf{safety}}$ and $\mathsf{Simplex}_{\mathsf{complex}}$ to be the hybrid automata corresponding to the safety and complex modes of Simplex. That is, $\mathsf{Simplex}_{\mathsf{safety}}$ and $\mathsf{Simplex}_{\mathsf{complex}}$ are each a hybrid automaton with variables $\{x_1, \ldots, x_n, c\}$. For a state $\mathbf{x} \in val(\{x_1, \ldots, x_n, c\})$, we refer to the values of the n continuous components by $\mathbf{x}.x_1, \ldots, \mathbf{x}.x_n$, or aggregately as $\mathbf{x}.x$, and the value of the clock as $\mathbf{x}.c$.

The unsafe states for the system are specified by a predicate U_x on the continuous state space, $\mathbb{R}^n$, and thus, the overall unsafe set is $U = U_x \times \mathbb{R}_{\geq 0}$.

We construct the discrete abstractions $\mathsf{AbsSimplex}_{\mathsf{safety}}$ and $\mathsf{AbsSimplex}_{\mathsf{complex}}$ of the given concrete $\mathsf{Simplex}_{\mathsf{safety}}$ and $\mathsf{Simplex}_{\mathsf{complex}}$ automata by quantizing the continuous state space. The construction technique for the $\mathsf{AbsSimplex}_{\mathsf{safety}}$ and $\mathsf{AbsSimplex}_{\mathsf{complex}}$ abstract discrete transition systems is analogous. Therefore, we now only refer to the construction of a $\mathsf{AbsSimplex}_{\mathsf{mode}}$ discrete transition system which abstracts a $\mathsf{Simplex}_{\mathsf{mode}}$ automaton, and assume the corresponding differential equations, referred to simply as F, are being used for the $mode$ under consideration.

A. Dynamics Restrictions

First, we place three restrictions on the mode's differential equations, F, which, in turn, restrict the underlying dynamics. Systems with dynamics that do not meet these restrictions are not applicable with our approach (we do not claim to be able to provide a discrete abstraction for every possible hybrid automaton). Later, in our case study in Section V, we show that these restrictions are not overbearing by applying the developed technique towards rollover prevention of autonomous off-road vehicles.

We first list the three restrictions on the differential equations, F, and then describe each one in more detail.

R1 For each continuous variable x_i, there exist upper and lower bounds x_{iu} and x_{il}, such that for all reachable states $\mathbf{v}$ of $\mathsf{Simplex}_{\mathsf{mode}}$, $\mathbf{v}.x_i \in [x_{il}, x_{iu}]$.

R2 For any continuous variable x_i, the right hand side of the (possibly nondeterministic) differential equation for $\dot{x}_i$ is expressible as a (possibly nondeterministic) function $f_i(\mathbf{v}.x)$. We assume that, for any product of half-open intervals $S \subseteq \mathbb{R}^n$, there exist computable bounds, $l_S, u_S \in \mathbb{R}$, on the derivatives such that, for all $\mathbf{v}.x \in S$, $f_i(\mathbf{v}.x) \in [l_S, u_S]$.

R3 The dependency graph of the continuous variables induced by the differential equations $\dot{x} = F(x)$ is acyclic, save for self-dependencies.

Many physical variables such as velocity, acceleration and temperature, can be associated with upper and lower bound values in the context of a given system. Such bounds can be used in place of x_{il} and x_{iu} in Restriction **R1**. This implies that our restricted class of $\mathsf{Simplex}_{\mathsf{mode}}$ models can not capture, for example, an unbounded position of a particle on the real line. This requirement is necessary for the constructed abstract discrete transition system to be of finite size, and therefore

[3]That is, not controlled by Simplex.

directly usable by a model checker to determine the Simplex guards.

The second restriction, **R2**, requires that, for every trajectory τ in $\mathsf{Simplex}_{\mathsf{mode}}$ that is within an arbitrary product of half-open intervals $S \subseteq \mathbb{R}^n$, it is feasible to determine bounds on the minimum and maximum rate of change (differential inclusions) of any continuous variable along τ. The simplest example of this is if the minimum and maximum possible rates of change of a variable remain constant with respect to time over all τ. For example, the rate of change of the angular velocity of a wheel is always bounded by the minimum and maximum angular acceleration that can be produced by the associated motor over all operating conditions (assuming there are no brakes). Such derivative bounds can also be derived for variables which change based on their own values, or the values of other variables. This would be the case if, for example, the rate of change of the angular velocity of a wheel was modeled as depending on the current angular velocity of the wheel. As we approach motor saturation (a high angular velocity), we can no longer increase the angular velocity as rapidly as we could when at rest. An example of a dependency on another variable would be if the rate of change of the angular velocity was modeled based on the pitch angle of the road on which we were driving. Symbolically, suppose x_1 evolves according to $\dot{x}_1 = f_1(x_1, x_2)$. Given a closed interval I_1 of values of x_1 and a closed interval I_2 for values of x_2, **R2** requires that we can compute u, l such that for all states $\mathbf{v}$ where $\mathbf{v}.x_1 \in I_1$ and $\mathbf{v}.x_2 \in I_2$, $f_1(\mathbf{v}.x_1, \mathbf{v}.x_2) \in [l, u]$. This restriction is necessary to be able to apply our algorithm for constructing the transitions of the abstract discrete transition system.

Restriction **R2** also limits the sorts of systems for which we can form discrete abstractions without introducing significant pessimism. Particularly, if we can determine the exact discrete state of the hybrid automaton $\mathsf{Simplex}_{\mathsf{mode}}$ which the system is in from only the values of $\mathbf{v}.x$, finding such bounds does not add significant pessimism into the abstract system we are constructing (the bounds from the states corresponding to the product of half-open intervals under consideration can be determined directly). However, if the hybrid automaton $\mathsf{Simplex}_{\mathsf{mode}}$ can be in multiple discrete states for a particular value of $\mathbf{v}.x$, the bounds must encompass bounds from all possible discrete states, which may introduce pessimism depending on the variation in the differential equations at the different states. The effect of this pessimism will be an abstract discrete transition system which may contain spurious transitions which do not reflect actual system dynamics. Since our guards will be determined based on the constructed discrete transition system, we may switch to the safety controller earlier than is necessary because of these spurious transitions.

Restriction **R3** forbids certain types of feedback interconnections in the underlying system dynamics. Particularly, when the change for each variable x_i over the change in time is defined as $\dot{x}_i = f_i(x_i^a, x_i^b, \ldots)$, we obtain a derivative dependency graph. Restriction **R3** forbids this dependency graph from having circular dependencies, except for self-dependencies. This means, for example, for two variables x_1 and x_2, dynamics like

$$\dot{x_1} = f_1(x_2) = x_2$$
$$\dot{x_2} = f_2(x_1) = -x_1$$

are not allowed, but dynamics like

$$\dot{x_1} = f_1(x_1, x_2) = x_1 + x_2$$
$$\dot{x_2} = f_2(x_2) = -x_2$$

are permitted. This is necessary to be able to apply our algorithm for constructing the transitions of the abstract discrete transition system, as will be shown in the Section IV-C.

B. Abstraction Function

When defining our abstraction function, which takes a concrete state in $\mathsf{Simplex}_{\mathsf{mode}}$ and maps it to an abstract state in $\mathsf{AbsSimplex}_{\mathsf{mode}}$, we first fix a collection of quantization parameters $q_1, \ldots, q_n \in \mathbb{R}_+$—one for each continuous variable. The choice of quantization parameters is a tunable trade off, where larger constants will lead to more pessimism in the associated discrete transition system (which means the decision module derived based on the discrete transition systems will be more pessimistic, possibly switching to the safety controller before it is absolutely necessary), and smaller constants lead to discrete transition systems with more discrete states (which increases the amount of time necessary for a model checker to iterate the system to determine the decision module's switching guards, as will be discussed later in Section IV-D).

A $\mathsf{Simplex}_{\mathsf{mode}}$ discrete transition system has set of variables $Y = \{y_1, \ldots, y_n\}$, where each y_i is integer-valued. For $\mathbf{y} \in val(Y)$, we refer to values of the n components by $\mathbf{y}.y_1, \ldots, \mathbf{y}.y_n$.

The quantization-based *abstraction function*, $\textsc{quantize} : \mathbb{R}^n \to val(Y)$, is defined as: for any $\mathbf{x} \in \mathbb{R}^n$, $\textsc{quantize}(\mathbf{x}) = \mathbf{y}$, where for each $y_i \in Y$, $\mathbf{y}.y_i = \lfloor \frac{\mathbf{v}.x_i}{q_i} \rfloor$. Notice that the explicit notion of clock is dropped when performing the abstract mapping. Instead, each transition in $\mathsf{AbsSimplex}_{\mathsf{mode}}$ corresponds to a series of trajectories and/or discrete transitions in the $\mathsf{Simplex}_{\mathsf{mode}}$ automaton, with total duration *at most* Δ (where Δ is, again, the Simplex controller iteration time parameter). The abstraction function defines the states and transitions of $\mathsf{AbsSimplex}_{\mathsf{mode}}$ according to Definition 3. An algorithmic way to construct these transitions will be discussed in the next subsection. For convenience, we also define two related functions, $\textsc{quantize-set}$ and $\textsc{quantize}^{-1}$. The function $\textsc{quantize-set} : \mathbb{R}^n \to 2^{val(Y)}$ is the extension of $\textsc{quantize}$ to larger sets of $\mathbb{R}^n$, while $\textsc{quantize}^{-1}$ is the *concretization map*, $\textsc{quantize}^{-1} : val(Y) \to 2^{\mathbb{R}^n}$, which is the inverse of $\textsc{quantize}$. The concretization of an abstract state gives us the corresponding product of half-open intervals in the continuous space, specifically,

$$\textsc{quantize}^{-1}(\mathbf{y}) = \prod_{i=1}^{n} \big[q_i * \mathbf{y}.y_i, q_i * (1 + \mathbf{y}.y_i)\big).$$

C. Algorithm for Constructing Discrete Transitions

Assumption **R1** implies that for each continuous variable x_i, we can restrict the type of x_i to the closed interval $[x_{il}, x_{iu}]$ of $\mathbb{R}$ without altering the behavior of the $\mathsf{Simplex}_{\mathsf{mode}}$ automaton model. Since each quantization parameter q_i for the corresponding variable x_i is strictly positive, it follows that the type (set of all possible values) of the abstract variable y_i is the *finite* set $\{\lfloor \frac{x_{il}}{q_i} \rfloor, \ldots, \lfloor \frac{x_{iu}}{q_i} \rfloor\}$.

Since there are a finite number of abstract discrete states for each variable, and there are a finite number of variables, there are a finite number of abstract discrete states in the aggregate system consisting of all the variables. Therefore, we construct the possible transitions of the abstract system by considering one abstract state $\mathbf{y}$, and repeat the procedure over the finite number of abstract system states.

Requirement **R3** rules out circular dependencies (except self-dependencies) in defining the derivatives of continuous variables within a trajectory. Therefore, ignoring self-dependencies, we can form a directed acyclic graph (DAG) of these dependencies and perform a topological sort. For the rest of the algorithm, we assume the variables, $x_1, x_2, \ldots, x_n$, are ordered according to this topological sort.

We now determine abstract transitions which correspond to trajectories of $\mathsf{Simplex}_{\mathsf{mode}}$ from a particular abstract state $\mathbf{y}$. We first determine the reachable abstract values for the y_1 variable, then the reachable abstract values for the y_2 variable, and so on. After this, we combine the cross product of the reachable values of all the variables to construct the possible transitions from state $\mathbf{y}$. The new goal, then, is determining the reachable abstract values for each variable from state $\mathbf{y}$.

Initially, we consider, x_1, which is has no non-self incoming edges in the derivative dependency DAG (since the variables are ordered according to the DAG's topological sort). Since x_1 has no non-self incoming edges, the bound function for its derivative, $\dot{x}_1$, is defined as either a constant, or a function of x_1. In both cases, by requirement **R2**, it is feasible to determine bounds on the minimum and maximum rate of change of x_1, the continuous variable, by considering a half-open interval corresponding to the current possible concrete states. This interval can be obtained by considering the dimension corresponding to y_1 of the current possible concrete states, $\text{QUANTIZE}^{-1}(\mathbf{y})_1$.

However, since the bounds on $\dot{x_1}$ are only valid within the interval, the bounds may change if the variable enters another interval before Δ time passes (recall that we are trying the determine the reachable values of the variable in up to Δ time). Consider the situation shown in Figure 3. Here, an initial state corresponding to the grey interval is evaluated for reachability. If we were to only consider the derivative bounds at the current interval, $\dot{x} = [-0.5, 0.5]$, we would not consider the state corresponding to the interval [3, 4) as reachable in one time step, whereas in actuality if the value of the variable enters the state corresponding to the interval [2, 3), our maximum derivative increases and we may, in fact, reach the interval [2, 3).

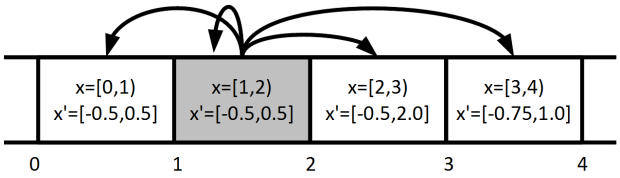

Fig. 3. A state (grey) interval within a self-dependent variable is evaluated for abstract discrete reachability within one time step. The updated product of half-open intervals in the associated dimension would reflect the concrete values corresponding to the reachable intervals, in this case [0,4).

In general, to determine the minimum-valued abstract state we can reach in up to Δ time, we start at the lower bound of the current state's interval and iterate the discrete states using the minimum derivative until we exhaust Δ time (or the minimum derivative becomes nonnegative), using new minimum derivatives as we enter new intervals. To determine the maximum state we can reach after up to Δ time, we start at the upper bound of the current state's interval and iterate the discrete states using the maximum derivative until we exhaust Δ time (or the maximum derivative becomes nonpositive), again, using new maximum derivatives as we enter new intervals. The reachable values of the variable under consideration in up to Δ time, then, are bounded between the two resultant states. We therefore have determined the reachable values for this variable in up to Δ time.

After determining the reachable values for the first variable in the topological sorted order, we update the product of half-open intervals for computing derivative bounds to reflect the values of x_1 reachable in up to Δ time. Thus, when computing the derivative bounds of any variables whose derivatives depend on the value of x_1, all values of x_1 reachable in up to Δ time will be considered. The algorithm then continues iterating each variable in the topologically sorted order. Thus, for evaluating the reachable values for an arbitrary variable y_j, we will always have the intervals of reachable values in up to Δ time for every dependent variable, except possibly y_j. We can then determine the reachable values of variable y_j by using the iterative procedure outlined above, starting with the variable interval corresponding to y_j, $\text{QUANTIZE}^{-1}(\mathbf{y})_j$.

After we determine the abstract states reachable for each variable starting from $\mathbf{y}$, the cross product of the reachable values for each variable encodes the possible discrete transitions corresponding to all possible continuous evolution of the variables in up to Δ time. If we repeat this procedure for every discrete state in the system, we will get a complete discrete transition system, $\mathsf{AbsSimplex}_{\mathsf{mode}}$, which is an abstraction of the continuous automaton, $\mathsf{Simplex}_{\mathsf{mode}}$.

The complete algorithm is outlined below. The algorithm uses the MinReach function (and MaxReach, which is similar) which is also defined. Here, $\mathcal{D}$ refers to the set of discrete transitions. Prior to the start of the algorithm, the variables are ordered according to the topological sort of the graph formed by their differential equation dependencies, ignoring self-dependencies.

```
ConstructTransitions()
{
  for each y ∈ val(Y)
    S ← QUANTIZE^{-1}(y)
    for each y_i ∈ y in the topologically sorted order
      reach_min ← MinReach(Δ, S, q_i * y.y_i, i)
      reach_max ← MaxReach(Δ, S, q_i * (1 + y.y_i), i))
      S ← S_1 × ... × S_{i-1} × [ reach_min, reach_max] × S_{i+1} × ... × S_n
    for each y' ∈ QUANTIZE-SET(S)
      D ← D ∪ {(y, Time, y')}
}
```

```
Value MinReach(Time t, Subset S, Value x, DimensionNumber i)
{
  S ← S_1 × ... × S_{i-1} × [q_i * ⌊x/q_i⌋, q_i * (1 + ⌊x/q_i⌋)] × ... × S_n

  [l, u] ← DerivativeBounds(i,S)

  if l ≥ 0 % derivative is positive, done
    return q_i * ⌊x/q_i⌋
  else if t + l * q_i < 0 % time expires before reaching next interval, done
    return q_i * ⌊x/q_i⌋
  else
    return MinReach(t + l * q_i, S, q_i * (⌊x/q_i⌋ − 1), i) % consider next interval
}
```

Here, DerivativeBounds is the function which returns the differential inclusions, mandated by Requirement **R2**.

Notice that, depending on the Simplex time parameter, Δ, and the quantum parameters for each variable, $q_1, q_2, \ldots, q_n$, the result of this process may have some spurious abstract transitions which are not in the concrete, continuous system. However, it is a simulation of the concrete system, in that any concrete combination of trajectories and discrete changes in the concrete system, $\mathsf{Simplex}_{\mathsf{mode}}$, will have a corresponding set of transitions in the abstract system, $\mathsf{AbsSimplex}_{\mathsf{mode}}$. By applying this technique for the two Simplex modes, you obtain two discrete transition systems.

D. Finding Guards from Discrete Abstractions

Having constructed the discrete abstractions for the individual modes of Simplex, $\mathsf{AbsSimplex}_{\mathsf{safety}} \triangleq \mathcal{S}$ and $\mathsf{AbsSimplex}_{\mathsf{complex}} \triangleq \mathcal{C}$, we now describe the verifiable construction of the guards G_c and G_s that constitute the switching logic. Suppose that the unsafe set of states is $\mathcal{U}$. A safe switching predicate G_c can be computed as:

$$G_c = \text{QUANTIZE}^{-1}(\mathsf{Prev}_{\mathcal{C}}(\mathsf{BackReach}_{\mathcal{S}}(\mathcal{U}))).$$

In other words, we consider all the set B of states that are backwards reachable from $\mathcal{U}$ using the discretized safety controller. Then we consider the set C of states that can reach B using a single step of the complex controller and set G_c as the set of concrete states corresponding to C. At a high level, states in G_c are ones where, if we use the complex controller for one control iteration and then switch to the safety controller, the system state may still eventually enter the unsafe region. Therefore, during execution, if the current state is in G_c, we must use the safety controller immediately. By starting in an initial state that is not in G_c and following this strategy, the system model will never enter an unsafe state.

In the next section, we illustrate using a model checker to perform this procedure within an case study.

V. Case Study

We now describe a case study which aims to formally provide property guarantees for autonomous off-road vehicles. Although we simultaneously investigated several integrity and performance properties [10], here we discuss only off-road vehicle rollover prevention. After a brief problem description in Section V-A, we describe the discrete model generation in Section V-B, which is based on the theory developed earlier in Section IV-C. Then, we use a model checker to determine the Simplex automaton guards, as well as verify the correctness of the aggregate system. Finally, we discuss creating a verified and precisely-timed implementation of the decision module based on these guards in Section V-C.

Throughout the case study, we work with the Maude [11] model checker. Maude is a high-performance rewriting framework which uses *rewrite equations* and *rewrite rules* to codify models, both of which can be specified conditionally. While rewrite equations perform serial and deterministic computations, rewrite rules can be used to encode parallelism and nondeterminism. Since our system derivatives are bounded by closed intervals, nondeterministic rewrite rules are used to discretely encode the nondeterministic system dynamics. The Maude engine can then exhaustively model check and verify properties about all possible executions of the system.

A. Rollover Description

Although there is no operator inside to injure, autonomous vehicle rollovers may cause damage to vehicles and damage to property. The high-level system may be designed to avoid high-risk situations such as high-sloping terrain, sharp turning, and high speeds, but since automatic controllers on off-road vehicles run sophisticated algorithms, integrating proprietary with commercially-available software, often using COTS hardware, formally verifying rollover avoidance is difficult. A Simplex implementation for rollover prevention is therefore a simpler system to validate.

We must describe the rollover property formally in order to verify that a model of an autonomous off-road vehicle never experiences rollover. We pessimistically state that the rollover property is violated whenever one of the wheels of the vehicle lifts off the ground, which is never part of normal off-road vehicle operation. The relevant physical constants and variables are shown in Figure 4. The equation to determine if the rollover property is violated is:

$$g < \frac{v^2 * \sin(\beta)}{\text{WB}} * \frac{\sin\left(\tan^{-1}\left(\frac{2\text{H}}{\text{TR}}\right)\right)}{\cos\left(\tan^{-1}\left(\frac{2\text{H}}{\text{TR}}\right) + \alpha\right)}$$

g: gravitational acceleration
WB: vehicle wheel base
2H: twice the vehicle's height of center of gravity
TR: vehicle track width
v: vehicle velocity
β: vehicle steering angle
α: slope terrain angle

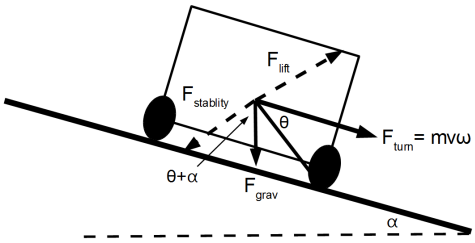

Fig. 4. The rollover condition is violated whenever F_{lift} exceeds $F_{\text{stability}}$. Here, ω is the angular velocity, calculated as $v * \sin(\beta)/\text{WB}$. The angle θ is computed as $\tan^{-1}(2\text{H}/\text{TR})$.

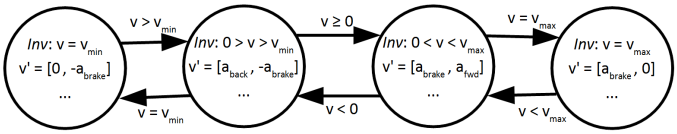

Fig. 5. This hybrid automaton captures the behavior of the velocity variable, v, in our rollover model. A complete system model would also include the terrain angle α and the steering angle β. Here, v_{min} is the minimum velocity, v_{max} is the maximum velocity, a_{fwd} is the maximum acceleration when moving forward, a_{back} is the minimum acceleration when moving backwards, and a_{break} is the minimum acceleration due to breaking when moving forwards, which is the opposite of the maximum acceleration due to breaking when moving backwards.

There is also a symmetric rollover case where $\alpha < 0$ and $\beta < 0$. Additionally, from this equation we can infer the behavior of the safety controller. To avoid rollover, the safety controller should reduce the steering angle β, and reduce the velocity v, which matches our intuition about rollover.

B. Discrete Model Generation

Once we know relevant variables for the property we are interested in verifying, we need to construct the hybrid model of the tractor system. The variables relevant to rollover are the angle of the terrain α, the steering angle β, and the velocity of the tractor v. Within our case study, we constructed a parameterized model generator, which takes as input about 30 parameters about the off-road vehicle system (such as the absolute maximum and minimum velocity, the discretization quantum for each variable, the rates of change of the steering angle, the track length and the wheel base), and outputs a Maude model which encodes the associated discrete abstract transition system. Our model consists of three variables, the slope angle α, the steering angle β, and the velocity v. The rates of change, $\dot{\alpha}$, $\dot{\beta}$, and $\dot{v}$, depend only on their own values, and $\dot{v}$ has different differential equation ranges depending on the direction of travel, as shown in Figure 5. As previously discussed in Section IV-A dealing with restriction **R2**, such an automaton for our velocity dynamics can be naturally modeled with our approach by a discrete abstraction, and does not introduce excessive pessimism because of the bounds computation. For each specific value of v, there is exactly one corresponding discrete state in the hybrid automaton dynamics description.

Our model generator runs the algorithm described in Section IV-C and outputs Maude modules encoding a discrete transition system for $\mathsf{AbsSimplex}_{\mathsf{complex}}$, which simulates the possible dynamics for the off-road vehicle system when the complex controller is used.

Within a single execution of the Maude model of the complex controller, a nondeterministic execution takes place since, at every point in time, there are many rewrite rules which can be applied to the system. This is expected, since we do not know which possible action the complex controller will take and there are always several options. More importantly, when the Maude engine performs model checking on the system, it iterates all possible rewrite rules exhaustively, allowing us to verify properties for all possible behaviors of the complex controller. For the safety controller, where the vehicle slows down and reduces the steering angle, the model generator creates rewrite equations describing the maximum rate at which we can slow down and the maximum rate at which we can straighten the wheels, based on the input parameters. The safety controller's dynamics, and the corresponding discrete transition system $\mathsf{AbsSimplex}_{\mathsf{safety}}$, then, are described by transitions where the slope angle, α, is allowed to change nondeterministically, but the velocity, v, reduces at the maximum allowed rate and the steering angle, β decreases at the maximum allowed rate.

C. Guards from the Model Checker

After we have Maude models corresponding to $\mathsf{AbsSimplex}_{\mathsf{complex}}$ and $\mathsf{AbsSimplex}_{\mathsf{safety}}$, we can use the model checker to determine the guards for our system. We primarily focus on determining the guard G_c from Figure 2, which is the set of states where we should switch from the complex controller to the safety controller. The guard for the reverse transition, from the safety controller to the complex controller, can use $G_s = \overline{G_c}$, or, more likely, $G_s \subset \overline{G_c}$, if a hysteresis is desired to prevent frequent controller switching.

To determine this guard set, we use the discrete transition systems for each $mode$ and the model checker to perform state-space searches. We can use the technique from Section IV-D and have the model checker output the states in G_c.

The behavior of the decision module, then, is to take the quantized current state of the system, and check if it belongs to the state set output by the model checker. If it does, the safety controller must be used. If it does not, the complex controller can actuate the system. The last step is to encode the state set G_c output by the model checker into a form that can be checked online, while the system is running.

D. Verifiable Implementation

To provide a verifiable implementation, we manually constructed linear bounds which captured all the states in G_c. After we have determined the bounds on the states where the safety controller should be used, we create the implementation of the decision module. To aid in this step, we have created formal semantics for a subset of VHDL (a hardware description language) within Maude [12], particularly enough of the language to be able to perform bounds checking.

The first step is to create the decision logic implementation in the VHDL semantics syntax within Maude. This code is executable within the Maude model checker (because the formal semantics are defined within Maude), but has a direct correspondence with a VHDL module. Using the Maude-based implementation along with the semantics, we can perform a state-space search on the composite system which will now include the discrete $mode$ transitions determined by the guards, verifying that if we use the bounds as the switching strategy, the system will never enter a rollover state.

The second step, is to take the Maude-based implementation and translate it a VHDL module. This is a straightforward step since there is a direct correspondence between the Maude-based VHDL code and regular VHDL code. We have created a program which does this translation automatically. The output of the program, then, is a VHDL module which captures the behavior of the decision module which was formally checked by Maude to result in a system where rollover does not occur. The VHDL code can then be put through a hardware synthesis tool and executed on a Field Programmable Gate Array (FPGA), or turned into an Application-Specific Integrated Circuit (ASIC).

The advantage of using VHDL as a target language is twofold. First, from a real-time perspective, hardware logic is extremely predictable. Our VHDL Maude semantics include a cycle counter which can be used in conjunction with the model checker to determine an upper bound on the number of clock cycles used to perform the decision module logic (by performing a search for a state where the cycle counter is greater than some value and seeing no matching reachable states). The number of cycles, when combined with a clock frequency, gives us a worst-case execution time. The hardware synthesis tools then check that timing constraints of the VHDL module are met (that the hardware logic can actually execute at the desired clock frequency). The second advantage of targeting VHDL is that performing Simplex at a lower level (in dedicated hardware) has previously been shown to be able to capture more potential errors [5]. By using the lower-level, System-Level Simplex Architecture, our design is more robust.

Although currently done manually, we plan to automate the task of creating the bounds on the states where the safety controller should be used, by performing an n-dimensional convex hull construction. Then, the bounds could be automatically created in the Maude-based VHDL from the convex hull, verified by Maude, and translated into a synthesizable VHDL module. All these steps could be done without any user input, reducing the likelihood of errors.

VI. Related Work

Finding finite state abstractions for hybrid automata is a problem of fundamental importance in design and verification of cyberphysical systems. This is because the availability of a *bisimilar*, finite-state abstraction $\mathcal{B}$ for a hybrid system $\mathcal{A}$, makes it possible to design and verify $\mathcal{A}$ by solving similar problems for $\mathcal{B}$. This is desirable because the latter problem can be attacked with powerful model checking tools, such as Maude [11] or SMV [13]. Techniques for abstracting hybrid automata have been sought after since the inception of the field, and it was quickly found that finite-state, bisimilar abstractions may not exist for general hybrid systems [14], [15], [16]. Nevertheless, restricted classes of hybrid automata have been identified for which such abstractions exist and can be algorithmically constructed [17], [18]. Unfortunately, the classes of hybrid automata that arise from the types of vehicular systems we are interested, generally do not fall within these restricted classes.

Apart from bisimulation-based, finite-state abstractions, ordinary simulation-based abstractions have been used [19], [20] for deductive verification of hybrid systems. Most of these techniques involve significant manual work in finding the simulation relation that relates the concrete hybrid automaton $\mathcal{A}$ with the (simpler) abstraction $\mathcal{B}$.

Several related notions of abstractions based on approximate bisimulation relations have been proposed [21], [22], [23]. One work particularly relevant to our paper [23] shares a similar goal—that of constructing finite abstractions for designing (or synthesizing) controllers with certain properties. One difference, however, is that we work with traditional simulation relations that give a partition of the state space, instead of approximate simulations which give a covering of the continuous state space.

The general problem of synthesizing switching controllers that are correct by construction has been looked at earlier, mostly for linear and piece-wise affine systems [24], [25]. To the best of our knowledge, however, the existing work does not address differential inclusions and periodically-switched controllers.

The Simplex Architecture [2] has previously been used to verify properties of various systems. However, in terms of verifiable Simplex application, previous approaches required determining a Lyapunov function for continuous dynamics, or exhaustively checking a formal model for discrete dynamics [5]. Our approach performs a simulation of the dynamics in an abstract discrete transition system, which does not require finding a Lyapunov function, while still permitting a restricted class of hybrid dynamics. This allows us to prove properties of systems expressible within this class of hybrid automata, which was unavailable previously. The main drawback of our approach, however, is that we may introduce pessimism while constructing a system's corresponding discrete transition system.

Instead of Simplex, other approaches for improved software robustness can be applied such as N-version programming [26], where multiple versions of software are created independently from the same specification, and the output is obtained through voting. This approach suffers from the lack of statistical independence of programming errors [27], [28], as well as, for a constant amount of development effort, actually being less reliable than focusing on a single version over a wide range of parameter values [2].

VII. Conclusions

In this paper we have presented a methodology for designing the decision module (or the switching logic) of a Simplex-based supervisory control system using ideas from the theory from hybrid system verification. Specifically, first, a discrete-state abstraction of the different modes of the Simplex hybrid automaton is constructed by overapproximating the dynamics with a discrete transition system. Using a model checker on these discrete transition systems to compute the backwards reachable set from the unsafe set, the guard predicate for switching from the complex controller to the safety controller is obtained. Of course, there is a trade-off between tractability of model checking and the conservativeness of the resulting switching logic. The coarser the discrete abstraction, the faster the model checking step, but also, the more conservative the switching logic (i.e., the safety controller takes over more aggressively). Exploring this trade-off will be the subject of our future investigations.

We have also presented an application of the methodology to a case study in designing the supervisory controller for an autonomous off-road vehicle. In this context, we have discussed faithfully translating the computed switching logic to a hardware description language which can be immediately synthesized and executed on an FPGA. Thus, we have developed a workflow for reliable embedded system design, from a high-level design using hybrid automata abstractions and model checking, to a verified correct and timely implementation in hardware.

Acknowledgment

This material is based upon work supported by John Deere under Award No. UIUC-CS-DEERE RPS #19. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the supporting company.

References

[1] RTCA/DO-178B, "Software considerations in airborne systems and equipment certification," www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgAdvisoryCircular.nsf/0/50bfe03b65af9ea3862569d100733174/$FILE/AC25.1309-1A.pdf, 1992.

[2] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, no. 4, pp. 20–28, 2001.

[3] E. D. Ferreira and B. H. Krogh, "Switching controllers based on neural network estimates of stability regions and controller performance," in *Lecture Notes on Computer Science, Special Issue: Hybrid Systems VI*. Springer Verlag, 1999.

[4] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 400–412.

[5] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 99–107.

[6] D. Seto, E. Ferreira, and T. F. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," Technical Report Cmu/ sei-99-Tr-020.

[7] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Research and Development*, vol. 6, pp. 200–209, 1962.

[8] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005, also available as Technical Report MIT-LCS-TR-917.

[9] T. Wongpiromsarn, S. Mitra, R. M. Murray, and A. G. Lamperski, "Periodically controlled hybrid systems," in *HSCC*, ser. Lecture Notes in Computer Science, R. Majumdar and P. Tabuada, Eds., vol. 5469. Springer, 2009, pp. 396–410.

[10] S. Bak, "Industrial application of the system-level simplex architecture for real-time embedded system safety," Master's Thesis, University of Illinois at Urbana-Champaign, 2009.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, vol. 4350. Springer, 2007.

[12] S. Bak, "Verifiable vhdl generation with vmaude," www.cs.uiuc.edu/homes/ sbak2/ vmaude/ vmaude.html, 2009.

[13] C. M. University, "Symbolic Model Verifier," www.cs.cmu.edu/ modelcheck/ smv.html, 2009.

[14] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *ACM Symposium on Theory of Computing*, 1995, pp. 373–382.

[15] A. Tiwari and G. Khanna, "Series of abstractions for hybrid automata." in *5th International Workshop on Hybrid Systems: Computation and Control,*, ser. LNCS, C. Tomlin and M. R. Greenstreet, Eds., vol. 2289. Springer, 2002, pp. 465–478.

[16] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," in *Proceedings of the IEEE*, 2000.

[17] V. Vladimerou, P. Prabhakar, M. Viswanathan, and G. Dullerud, "Stormed hybrid systems," in *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 136–147.

[18] G. Lafferriere, G. J. Pappas, and S. Yovine, "A new class of decidable hybrid systems," in *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*. London, UK: Springer-Verlag, 1999, pp. 137–151.

[19] N. Lynch, "A three-level analysis of a simple acceleration maneuver, with uncertainties," in *Proceedings of the Third AMAST Workshop on Real-Time Systems*. Salt Lake City, Utah: World Scientific Publishing Company, March 1996, pp. 1–22.

[20] S. Mitra, D. Liberzon, and N. Lynch, "Verifying average dwell time of hybrid systems," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 1, pp. 1–37, 2008.

[21] A. Girard and G. J. Pappas, "Approximation metrics for discrete and continuous systems," in *IEEE Transactions on Automatic Control*, 2005.

[22] A. Girard, A. A. Julius, and G. J. Pappas, "Approximate simulation relations for hybrid systems," in *IFAC Analysis and Design of Hybrid Systems*, Alghero, Italy, June 2006.

[23] P. Tabuada, "Approximate simulation relations and finite abstractions of quantized control systems," in *HSCC*, ser. Lecture Notes in Computer Science, A. Bemporad, A. Bicchi, and G. C. Buttazzo, Eds., vol. 4416. Springer, 2007, pp. 529–542.

[24] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli, "Effective synthesis of switching controllers for linear systems," in *Proceedings of the IEEE*, 2000, pp. 1011–1025.

[25] C. Belta, L. Habets, and V. Kumar, "Control of multi-affine systems on rectangles with applications to hybrid biomolecular networks," in *41st IEEE Conference on Decision and Control*, Las Vegas, NV, 2002.

[26] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during program execution," in *COMPSAC 77*, 1977, pp. 149–155.

[27] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering*, vol. 12, no. 1, pp. 96–109, 1986.

[28] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an n-version software experiment," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 238–247, 1990.