Chapter 1

Introduction

1.1 Motivation

Autonomous robotic systems have many applications, such as planetary exploration missions like the Titan Aerobot proposal (Figure 1.1) [1], or high-risk reconnaissance or security duties, which may be applications of the DARPA Urban Challenge vehicles (Figure 1.2) [2]. As the missions that the robots take on become more complex, so do the robots' control systems. For the high-risk observation and exploration missions, the autonomous system must be prepared to encounter a dynamic environment that must be observed using some set of sensors. The dynamic environments and sensor suites increase the complexity of autonomous systems and increase the number of ways that robots can fail. Poor characterizations of the capabilities of the robot and incomplete models of the environment have caused the downfall of many autonomous systems. An example is NASA's DART (Demonstration of Autonomous Rendezvous Technology) spacecraft, which crashed into its rendezvous target. The collision avoidance software on the spacecraft failed to function because of the discrepancy between the spacecraft's actual and estimated positions [3]. Another example is Caltech's entry into the 2005 DARPA Grand Challenge competition. A failed LADAR unit and a degraded GPS measurement compromised the autonomous vehicle's knowledge of its position and caused it to swerve off the road into some cement barriers [4]. In both cases, the health of the sensors that were contributing data to the position estimation was not considered by the control system, causing a system failure.

The main contributor to the complexity of a control system is often its fault protection. Many times, the necessary fault detection, isolation and recovery software for autonomous robotic systems is cumbersome and added on as failure cases are encountered in simulation. There is a need for a systematic way to incorporate fault tolerance in autonomous robotic control systems in all stages of

Figure 1.1: The Titan Aerobot model

Figure 1.2: Alice, Caltech's entry in the DARPA Urban Challenge

system design. One way to increase the fault tolerance of a system is to reduce its autonomy. For example, in traditional robotic space missions that use command sequence-based control systems, the most common complex fault response that is used for all but the most critical times in the mission is called safing [5]. Safing is a sequence of commands that a system executes to put the robot into a 'safe' configuration. Mission Control is then responsible for deciding what the fault response should be. However, there are times when human intervention is expensive or even impossible, such as the time-critical entry, descent and landing sequence of Mars exploration robots; then, an autonomous fault tolerant control system is necessary.

One way to design a fault tolerant autonomous system is to create a flexible control system that can reconfigure itself in the presence of faults. However, reconfigurability adds complexity that could reduce the system's effective fault tolerance. The fault tolerant control system must be tested to ensure that the system performs in a safe manner whenever a fault occurs, and it must be tested to ensure that there is a control tactic to account for all possible faults and failures. Typical validation testing using case studies and simulations is not thorough enough to guarantee fault tolerant behavior. A more rigorous type of testing is needed.

The safety verification of a complex control system can prove that the system will perform in a safe and expected way upon any combination of failures [6]. The ability to reach certain unsafe

states can be tested; if these states are not reachable, the control system is considered to be verified with respect to the unsafe states. Some examples of unsafe states that could be analyzed include irrecoverable low power states or collisions with sensed objects. There are many methods available to verify a control system; symbolic model checkers can partition and search the state space of many types of simple deterministic systems, while other methods can determine an upper bound on the probability of failure of systems that include uncertainty. In this dissertation, three methods of verifying complex, goal-based, fault tolerant control systems both with and without uncertainty will be presented along with the verification of two significant examples.

1.2 Fault Tolerant Control

Fault tolerance describes a system's ability to continue functioning, possibly in a degraded manner, upon some fault or failure in the system. Fault tolerance should be considered in a system's design phase. The first important step in having a fault tolerant control system is being able to detect faults and failures in the system. Fault detection and isolation techniques come in many forms, but all involve the estimation of system or environment states that are important to the health of the control system. The most popular estimators in robotics are the many variants of Kalman filters and extended Kalman filters; however, their performance can depend heavily on the quality of the models provided. One way to avoid this dependence on uncertain model parameters is to introduce a method for automatically learning noise parameters [7]; another tactic is to use multiple parallel Kalman filters to capture the modeled behavior in each fault mode [8], assuming that the fault modes are finite and known. When this is not the case, it may be possible to use a model-based diagnosis technique with the ability to handle unknown modes, such as a partial filter formulation that is based on extended Kalman filters [9]. Other methods reduce or eliminate the dependence of the fault estimation on the models of the systems by using particle filters [10] or by using a behavior-based approach in which temporal fuzzy logic accounts for noise and uncertainty in the autonomous system [11].

Once the fault has been identified, the control system needs to utilize that knowledge. The notion that fault tolerance should be integrated into a control system from the initial design is a popular one [12]. Several fault tolerant control architectures for autonomous systems have been developed in which the control effort is layered to deal with faults on different levels, including low levels of hardware control and high levels of supervisory control, such as those in Ferrell [13], Visinsky et al.

[14], and Lueth and Laengle [15]. The fault tolerant control architecture ALLIANCE is a behavior-based control system for multi-robot cooperative tasks [16]. In ALLIANCE, the distributed control system re-allocates tasks between robots in response to failures. Although many fault tolerant control systems achieve reconfigurability, few actually change the control tactic given to the system. The system described in Diao and Passino [17] uses adaptive neural/fuzzy control to reconfigure the control system in the presence of detected faults, and another described in Zhang and Jiang [18] reconfigures both the control system design and the inputs to the control system, though neither adjusts the intent of the commands in response to failures.

The control decisions of fault tolerant systems must depend on the current state information. Model-based control ensures that these systems have all the state information needed to make good control decisions and the models are used to inform the system of which control tactic to use. Reactive, model-based programming languages have been developed [19] and applied to NASA's Deep Space One probe [20] and Mars exploration rovers [21], including the time-critical entry, descent and landing sequence [22]. As the control system becomes more state- and model-based, traditional command sequences become too rigid. Several control architectures have been designed to accommodate behavior-based [23] or layered robotic control systems [24].

A control software architecture developed at the Jet Propulsion Laboratory uses state models and reconfigurable goal-based control programs for the control of autonomous systems [25]. Mission Data System (MDS) is based on a systems engineering concept called State Analysis [26]. Using MDS, systems are controlled by networks of goals, which directly express intent as constraints on physical states over time. By encoding the intent of the robot's actions, MDS has naturally allowed more fault response options to be autonomously explored by the control system [27]. Unlike the traditional command sequences used to control robotic space missions, goal networks allow for branching in the execution plan at the cost of added complexity. The complex branching nature of goal networks make control system verification necessary. The control programs in this dissertation are modeled after the MDS architecture, which will be more fully described in Chapter 2.

1.3 Control System Verification

Verification is a technique to prove the correctness of a control system with respect to a specific property using formal methods. Two of the most popular verification techniques are theorem proving and model checking [6]. Theorem proving involves using the formal description of the system,

which defines sets of axioms and inference rules, to prove specific properties about the system. Several techniques and specification languages have been developed in order to facilitate the creation of proofs of safety properties. For example, inductive techniques can be used to prove safety properties of distributed systems when the subsystems have the same properties [28]. A guarded-command language called Computation and Control Language (CCL) uses the same set of tools to model, specify, analyze, and prove properties about the control system [29]. The design and verification of a distributed railway control system was accomplished by following the RAISE method [30], which translates mathematical specifications into implementable control processes. The original abstract algebraic specifications are then used to prove properties about the control system. Theorem proving has been partially automated; much input by a human designer is often necessary. Prototype Verification System (PVS), a general purpose automatic theorem prover, is one of the most popular [31].

The formal method used in this work is model checking. In model checking, the system is represented as a finite state machine or a set of hybrid automata and some specification, often expressed in temporal logic, is checked by efficiently searching the state space of the system. Model checking is nearly completely automatic, fast, and able to handle somewhat complex systems. Model checkers come in many varieties. The symbolic model checkers designed for systems with no continuous state space, such as Bebop, which verifies Boolean programs [32], Symbolic Model Verifier (SMV) and its variant NuSMV, which verify finite state machines against requirements written in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [33], and an algorithm for checking Mu-Calculus formulas using CTL requirements [34] all use Binary Decision Diagrams (BDDs) to symbolically represent the state space. These algorithms are capable of verifying systems with hundreds of discrete state variables. Another ω-automata based model checker, Spin, has been demonstrated on several complex distributed systems, including spacecraft control system requirements [35]. Other symbolic model checkers, such as Bounded Model Checker (BMC) [36], have moved away from BDDs and instead use propositional satisfiability (SAT) methods [37].

There is also a class of symbolic model checkers that can verify systems that have discrete and simple continuous states. Hybrid systems consist of discrete sets of continuous dynamics, called modes or locations, which are connected by transitions that can be guarded. When the continuous dynamics of these systems are sufficiently simple, it is possible to verify that the execution of the hybrid control system will not fall into an unsafe regime [38]. There are several symbolic model checking software packages available that can be used for the analysis of different variants of

hybrid systems and timed automata, including HyTech [39], UPPAAL [40], and VERITI [41]. Two symbolic model checkers are particularly applicable to the types of hybrid systems encountered in this work, HyTech and PHAVer [42]. PHAVer is a more capable extension of HyTech that is able to exactly verify linear hybrid systems with piecewise constant bounds on continuous state derivatives and is able to handle arbitrarily large numbers due to the use of the Parma Polyhedra Library. Unlike "pure" model checkers such as Spin [43] that exhaustively and directly search the entire state space, symbolic model checkers are able to abstract the state space, but they still suffer from state space explosion issues to a varying degree. Many state space reduction techniques and problem abstractions have been explored to try to minimize this problem [44], and while some of the reduction techniques are automated [45], most of the abstractions are not.

When analyzing a specific system, it is useful to be able to leverage a larger class of systems for verification tools and methods. The control programs may need to be transformed to an acceptable form by some suitable means. In general, it is important that the new converted representation of the control system is bisimilar to the original control system [46], that is, there exists a mapping that has the properties of soundness and completeness between the control system and its representation. Some examples of the creation of bisimulations are found in Tabuada and Pappas [47] and Girard and Pappas [48]. When the conversion of a control system is a bisimulation, it is guaranteed that if the converted representation can be verified, the original system is also verified. Several conversion algorithms exist for systems that do not conform to a model checking software's requirements. One such tool exists for the conversion of AgentSpeak, a reactive goal-based control language, into two languages: Promela, which is associated with the Spin model checker [43], and Java, for which Java Pathfinder 2 (JPF2) is a general purpose model checker [49]. A rule-based procedure to convert specifications into a Petri net model in order to verify the model is described in Suzuki et al. [50]. An automatic method to convert Model-based Programming Language (MPL) code into models that can be verified by the Livingstone fault diagnosis system exists [51]. A related procedure that converts between natural language and temporal logic specifications for use in the verification of systems in SMV has been explored [52].

To successfully verify an autonomous control system, it is necessary to plan for verification during the design phase. One approach is to use design for verification procedures to ensure that the resulting control systems have a structure that is conducive to verification. Many different styles of specification can be used to constrain the resulting system to be verifiable, including model and constraint-based specifications [53]. A model-based approach that is based on the Synchronization

Units Model uses Constraint Handling Rules to express the semantics of synchronization constraints in the specific middleware framework to be verified [54]. In another example, the concurrency controller design pattern was applied to an air traffic control autonomous separation software to allow for its verification by two different methods [55]. Another design for verification procedure was applied to Object Oriented Analysis to allow for a smooth interface to model-based verification techniques [56]. A more general approach involves following a set of rules that will result in a system design that is able to be decomposed for the verification effort [57, 58].

A more restrictive and rigorous approach that extends the design for verification concept is to create correct-by-design control programs. This has been done from Buchi automata on infinite words [59] or from specifications and requirements stated in a restricted subset of LTL [60]. The correct-by-design approach creates control programs that have guarantees of correctness; this removes the verification step. However, the structure that must be imposed on the control systems generally is very restrictive. The design for verification approach allows for less structure and more capable control systems, however, many of the current design for verification procedures are sets of complicated rules that the designer must follow and the procedures are only applicable to a specific design tool or method.

1.4 Stochastic Verification

The verification methods introduced in the last section do not account for noise and uncertainty in the systems being analyzed. Uncertainty makes the verification problem more difficult, though there are ways to verify uncertain or probabilistic systems. Reasoning about uncertain systems has driven the creation of probability-extended logics, like RTCTL, a realtime extension of CTL [61]. Other methods, such as using a stochastic concurrent constraint language to describe concurrent probabilistic systems [62] or using model checking ideas with fixed trajectories for analyzing stochastic "black box" systems [63], have been researched, but the most prevalent verification method is probabilistic model checking. In probabilistic model checking, an automatic algorithm determines if some specified property holds in a probabilistic system model [64]. These system models are generally derivatives of Markov models, such as continuous-time Markov chains [65], but timed probabilistic automata and stochastic hybrid models are also possible [66].

Stochastic hybrid models include uncertainty in the transitions of the hybrid automata as probabilistic transition conditions and include uncertainty in the continuous state evolution using stochas-

tic differential equations. Many methods to verify stochastic hybrid systems exist. For example, Prajna et al. [67] use barrier certificates to bound the upper limit of the probability of failure of the stochastic hybrid system and Kwiatkowska et al. [68] discuss a probabilistic symbolic model checking software called PRISM. A computational method that characterizes reachability and safety as a viscosity solution of a system of coupled Hamilton-Jacobi-Bellman equations analyzes stochastic hybrid systems by computing a solution based on discrete approximations [69]. Probabilistic reachability analysis techniques have been developed for controlled discrete-time stochastic hybrid systems [70, 71] and for large-scale stochastic hybrid systems using rare event estimation theory [72] and subset simulation [73]. When the stochastic hybrid systems become too large to reason about using the model checking and reachability analysis techniques, Markov Chain Monte Carlo techniques can be used to approximate a likelihood of system failure [74, 75].

1.5 Outline

The verification and analysis of goal-based control programs that are modeled after the goal networks used by the MDS control architecture are the topics of this dissertation. Chapter 2 gives some background information on MDS, hybrid systems, and stochastic hybrid systems. A bisimulation conversion procedure between goal networks and linear hybrid automata is presented in Chapter 3. The goal network conversion software based on the bisimulation is also introduced; this conversion allows the goal network to be verified by the PHAVer symbolic model checker, which is described briefly in this chapter. Chapter 4 describes a verification method for certain goal networks or hybrid systems that are designed in a rigorous way so that the transitions are completely state-based. A simple software design tool called the SBT Checker allows for the distributed and iterative design of goal networks that have the necessary properties to be verified by the InVeriant verification software. The InVeriant software is a model checker that exploits the structure of the rigorously designed goal network or hybrid system with state-based transitions to prove the reachability of unsafe conditions. Chapter 5 discusses a technique to compute the failure probability due to sensor-based state estimation uncertainty in hybrid systems that have been previously verified in the perfect knowledge case. Two significant example problems are verified in Chapter 6 using the methods introduced in the previous three chapters. Finally, Chapter 7 concludes the work and discusses directions for future research.