Safety Verification and Failure Analysis of Goal-Based Hybrid Control Systems

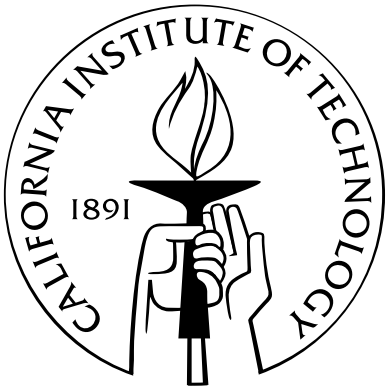
Thesis by

Julia M. B. Braman

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2009

(Defended May 27, 2009)

Acknowledgements

I would like to thank my advisor, Professor Richard Murray, for his support and encouragement. Richard is a great teacher and is extremely hard-working, accommodating, and enthusiastic about what he does, and all those qualities help make him a wonderful advisor.

I would like to thank Professors Jim Beck, Joel Burdick, and Mani Chandy for serving on my thesis committee. Having taken courses from all, I truly respect the enthusiasm each shows for his field.

There are several people that I must thank from the Jet Propulsion Laboratory. First, Mitch Ingham, Dave Wagner, and Kenny Meyer have been great resources, Mitch for all the technical discussions and brainstorming sessions, Dave for the tremendous amount of help in setting up and becoming proficient with MDS, and Kenny for keeping me connected with the MDS group at JPL. Bob Rasmussen and Dan Dvorak deserve special mention for their invaluable instruction on the intricacies of State Analysis and MDS. Many others on the MDS team have given feedback on the many presentations I have given, and I am grateful for their time and insight. Finally, some other JPL folks were kind enough to discuss ideas with me, most notably Alberto Elfes for the Titan aerobot discussions, and Gerard Holzmann and Rajeev Joshi for the Spin model checker discussions.

Last but not least, I would like to thank my family. My parents, Mark and Mary Ann Badger, fought for me when I could not fight for myself. My husband Kevin has been amazing during this journey and I am eternally grateful for his love and support. Thanks also goes to Cooper for sleeping through the night from a very young age.

Abstract

The success of complex autonomous robotic systems depends on the quality and correctness of their fault tolerant control systems. A goal-based approach to fault tolerant control, which is modeled after a software architecture developed at the Jet Propulsion Laboratory, uses networks of goals to control autonomous systems. The complex conditional branching of the control program makes safety verification necessary. Three novel verification methods are presented. In the first, goal networks are converted to linear hybrid automata via a bisimulation. The converted automata can then be verified against an unsafe set of conditions using an existing symbolic model checker such as PHAVer. Due to the complexity issues that result from this method, a design for verification software tool, the SBT Checker, was developed to create goal networks that have state-based transitions. Goal networks that have state-based transitions can be converted to hybrid automata whose locations' invariants contain all information necessary to determine the transitions between the locations. An original verification software called InVeriant can then be used to find unsafe locations of linear hybrid systems based on the locations' invariants and rate conditions, which are compared to the unsafe set of conditions. The reachability of the unsafe locations depends only on the reachability of the states of the state variables constrained in the locations' invariants from those state variables' initial conditions. In cases where this reachability condition is not trivially true, the software efficiently searches for a path to the unsafe locations using properties of the system. The third verification method is the calculation of the failure probability of the verified hybrid control system due to state estimation uncertainty, which is extremely important in autonomous systems that rely heavily on the state estimates made from sensor measurements. Finally, two significant example goal network control programs, one for a complex rover and another for a proposed aerobot mission to Titan, a moon of Saturn, are verified using the three techniques presented.

Contents

Nomenclature

β	Contribution set
χ	Uncertain state variable
Γ	Passive state space
$\mathcal{D}$	Set of passive state variables
$\mathcal{G}$	Set of goals in a goal network
$\mathcal{L}_{r,k}$	Set of executable branches of goals in $\mathcal{S}_{r,k}$
$\mathcal{S}_{r,k}$	Set of descendants of root goal $g_r^{0,0}$ in group $\mathcal{G}_k$
$\mathcal{U}$	Set of uncertain state variables
ν	Nominal path
Ω_k	Set of unsafe complete system states in group V_k
ϕ	Flow of an executable set of goals
Π	Set of failure paths
ψ_i	Flow equations for location v_i
ρ	Transition between executable sets of goals
Σ	Set of transition conditions in a hybrid system
τ	Transition condition in a hybrid system
Θ_k	Set of executable sets of goals in group $\mathcal{G}_k$
Υ'_k	Set of all consistent executable branch combinations

Ξ_k	Set of nominal complete system states in group V_k
ζ	Unsafe condition; set of unsafe constraints
A	Set of resets in a hybrid system
a_k	Initial failure probability of group V_k
B_k	Set of all contribution values in group V_k
c_k	Completion time for group V_k
E	Set of edges in a hybrid system
F_k	Set of Safing complete system states in group V_k
$g_n^{i_n,j_n}$	Goal
Q_k	Nominal transition probability matrix for group V_k
R_k	Set of root goals in group $\mathcal{G}_k$
S	Set of complete system states
T	Time point
t	Execution time
V	Set of locations in a hybrid system
W_k	Vector of initial nominal probabilities for group V_k
W_s	Failure probability
$W_{u,k}$	Failure transition probability vector for group V_k
X	Set of controlled state variables
Z	Unsafe set
k	Group number
n	Goal index

i_n Parent goal index

j_n Tactic number

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.

Chapter 2

Background Information

The goal-based control systems that are used in this work are modeled after the goal networks designed for the Mission Data System (MDS) control architecture. A useful way to represent these reconfigurable control programs is as hybrid systems because model checking is then possible. An introduction to State Analysis, the design methodology upon which MDS is based, and a notational introduction to linear hybrid systems are given in this chapter, along with a brief introduction of stochastic hybrid systems, which are useful for understanding the uncertainty analysis of linear hybrid systems.

2.1 State Analysis and Mission Data System

State Analysis is a systems engineering methodology that focuses on a state-based approach to the design of a system [26]. Models of state effects in the system that is under control are used for the estimation of state variables, control of the system, planning, and goal scheduling. State variables are representations of states or properties of the system that are controlled or that affect a controlled state [76]. Examples of state variables could include the position of a robot, the temperature of the environment, the health of a sensor, or the position of a switch. During the design process, the state variables are linked in a state effects diagram. The relationships between state variables, commands, and measurements denoted in this diagram are associated with corresponding state effects models.

Goals and goal elaborations are created based on the models. Goals are specific statements of intent used to control a system by constraining a state variable in time. Goals are elaborated from a parent goal based on the intent and type of goal, the state models, and several intuitive rules, as described in Ingham et al. [26]. A core concept of State Analysis is that the language used to design the control system should be nearly the same as the language used to implement the control system.

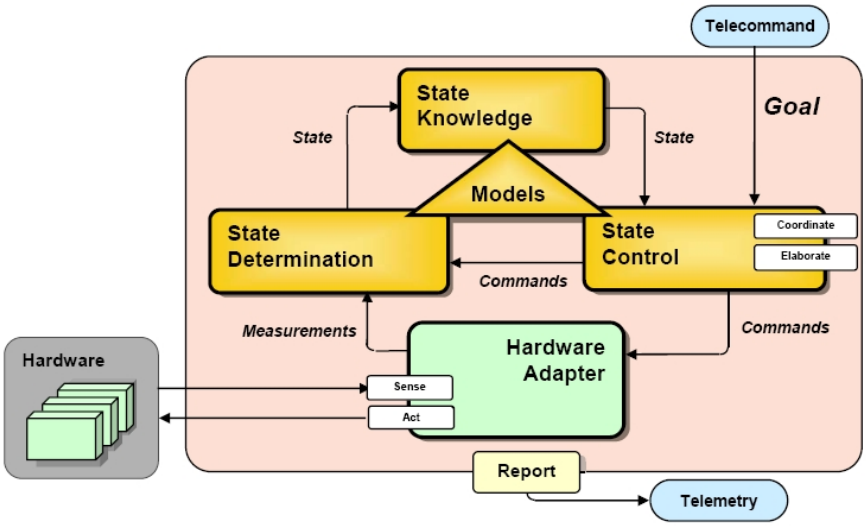


Figure 2.1: A depiction of the state and model-based architecture of the Mission Data System, from Dvorak [78]

Therefore, the software architecture, MDS, is closely related to State Analysis [77]. A depiction of the MDS control architecture is shown in Figure 2.1.

Goal networks in MDS replace command sequences in traditional control architectures as the input to the system. A goal network consists of a set of goals, a set of time points, and temporal constraints. A goal may cause other constraints to be elaborated on the same state variable and/or on other causally related state variables. The goals in the goal network and their elaborations are scheduled by the scheduler software component so that there are no conflicts in time, goal order or intent. Each scheduled goal is then achieved by the estimator or controller of the state variable that is constrained. Controlled goals cause some control action to be taken, either because of its constraint or because of a constraint in an elaborated child goal. Passive goals constrain the state of a state variable to be some specific value or set of values without an associated control action. Passive goals are used to choose between the tactics of a controlled goal; for example, the health values of redundant heaters are constrained such that the tactics of a temperature maintenance goal are achieved using the minimum power rate.

There are several types of controlled goal constraints [26]. *Macro goals* are goals that do not constrain any state variable, but elaborate controlled goals with constraints. A *maintenance goal*, mentioned above, uses control action to maintain the constrained value of a state variable. A maintenance goal on temperature, i.e., keep the temperature of an instrument between $15 - 20°$C, could elaborate control constraints on heater switches to achieve the temperature constraint. *Reset con-*

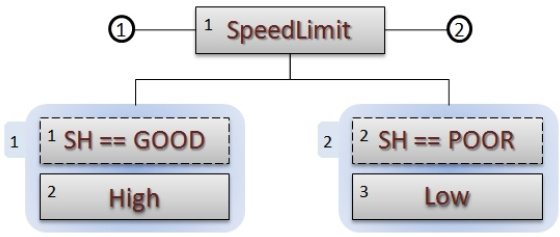


Figure 2.2: Goal tree example; circles are time points which bound the root goal. Rectangles are goal constraints (controlled goals have solid borders and passive goals have dashed borders; these types of goals are numbered independently in the upper left corner of the goal), and elaboration is signified by the tree structure. The parent goal's tactics are numbered in the shaded tabs.

straints command changes in discrete state variables (such as switches and states that have control modes). A *rate constraint* puts an upper or lower bound on the rate of some state variable. Finally, a controlled goal could have a *transition constraint*. This constraint drives a state variable from its current state to an end point; an example is a constraint to pan a camera to a given angle from some arbitrary starting value. The goal is achieved, or completed, when the state variable reaches the constrained value. For this reason, goals with this type of constraint are called *completion goals*.

Elaboration allows MDS more flexibility than control architectures based on command sequences. One example is fault tolerance. Re-elaboration of failed goals is an option if there are physical redundancies in the system, many ways to accomplish the same task, or degraded modes of operation that are acceptable for a task. The elaboration for a goal can include several pre-defined tactics. These tactics, or collections of concurrently elaborated and executed goals, are simply different ways to accomplish the intent of the goal, and passive goals constrain the conditions in which a tactic may be executed. A simple example of a speed limit goal and its elaborations (called a goal tree) is shown in Figure 2.2. The passive goals constrain the `SystemHealth` state variable (`SH`); the two tactics have controlled goals constraining different maximum speeds that could be allowed based on the `SystemHealth`'s value. Elaboration allows for many types and combinations of faults to be accommodated automatically by the control system [27].

The goal networks used for this work are closely related to the ones implemented by MDS. The biggest difference is how the goal networks are executed. In MDS, a goal network is initially scheduled and at that time, all parent goals elaborate only one tactic. The tactics chosen are based on specified elaboration order and/or the projections of the values of the constrained state variables. Projections of state variables are functions that predict the future value of a state variable based on its current (or initial) state, its state model and the goals constraining that state variable. Then,

as the goal network execution reaches an elaborated tactic, execution of that tactic begins with a check of the satisfaction of the constraint of the tactic based on the current values of the constrained state variables. If it is not achievable, or if the execution of that tactic is initially achievable but becomes unachievable during the execution, then the tactic will fail and re-elaboration will be triggered. During the re-elaboration, the entire goal network is rescheduled and re-elaborated with new projections while the failed tactic continues to execute. Once the rescheduling is complete, which could be several time steps later depending on the size of the goal network, the execution of the new tactic in the rescheduled goal network begins if it is still achievable.

Because these time delays in switching the execution tactics upon goal failure are problematic when one wishes to verify the behavior of a goal network, the execution of the goal networks in this work is assumed to be different. First, the goal network is scheduled initially, but elaboration does not occur in the same way. Instead of choosing one tactic per parent goal, essentially all possible goal network executions are elaborated. Then, tactics are chosen instantaneously based on the state variable values as the goal network execution reaches the parent goal. Upon goal failure, a new tactic of only the failed parent goal occurs, and the execution switches to the new tactic in the next time step. Because the goal network does not reschedule upon goal failure, some flexibility is lost, but in the structure that is imposed, a very useful property is gained. The time points are guaranteed to fire in the order that they are originally scheduled, which means the goal network is well-ordered. This property will be important to the verification process.

In a fault-tolerant control program with conditional branching, one execution branch or tactic is chosen over another based on the states of the system because it is the safest and best control tactic available. Therefore, continuing to execute a failed or incorrect tactic while the goal network re-elaborates is contrary to the intent of safety. For this reason, re-elaboration is assumed to happen in the same time step as goal failure. This is not an impossible assumption to execute; localized re-elaboration or total initial elaboration at scheduling could be implemented as a design choice.

Other new design choices are implemented for these goal networks. First, only controlled goals can elaborate other goals. If more than one tactic exists in a parent goal's elaborations, at least one goal in each tactic must be a passive goal. For convenience, controlled goals should not fail independently; instead, an accompanying passive goal would cause the tactic to fail. For example, assume there was a controlled goal constraining a heater switch to turn off, but that the heater was failed on. Instead of the controlled goal failing, an accompanying passive goal constraining the `HeaterSwitchHealth` state variable to be `GOOD` would fail, since its estimated state should be

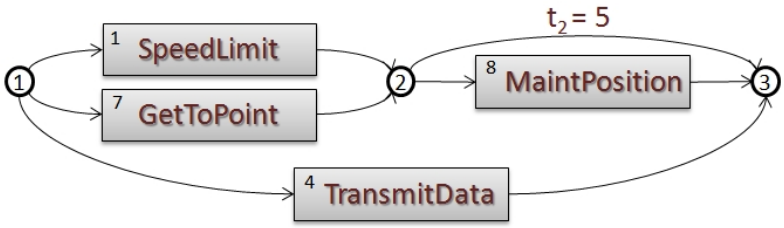


Figure 2.3: Goal network for simple example

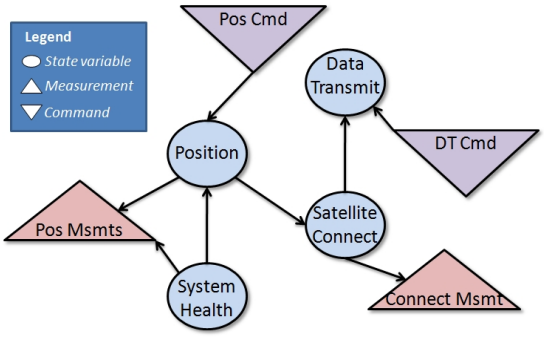


Figure 2.4: State effects diagram for simple example. Arrows indicate that the originating body has a modeled effect on the accepting body.

FAILON.

An example of a simple goal network that follows these design choices is shown in Figure 2.3. The goal network for this example has three time points, four root, or parent, goals, and it constrains four state variables, whose state effects diagram is shown in Figure 2.4. Two state variables are controlled, `Position` and `DataTransmission`; the former is a continuous state variable while the latter is discrete. The other two state variables are passive and have non-deterministic discrete models that are shown in Figure 2.5. Two of the root goals in the goal network do not have elaborations; however the goal trees for the SpeedLimit goal and the TransmitData goal are shown in Figure 2.6. The tactics shown in the goal trees contain passive goals on two state variables, `SystemHealth` and `SatelliteConnection`, that drive the choice of the tactics. The rest of the goals are controlled goals on the two controllable state variables and they cover several constraint types, which are shown in Table 2.1.

The overall task that this goal network is attempting to achieve is to drive a robot to a point while maintaining some safe velocity and transmitting data to a satellite. The execution of this goal network occurs as follows. The first time point fires, which starts the execution of the SpeedLimit, GetToPoint, and TransmitData goals. In addition, the SpeedLimit and TransmitData goals each elaborate one tactic based on the estimated states of the `SystemHealth` and `SatelliteConnection`

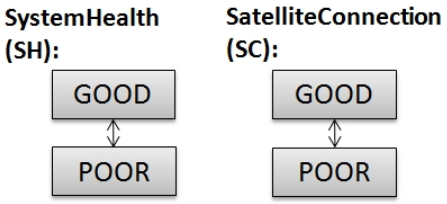


Figure 2.5: Non-deterministic models for the example passive state variables

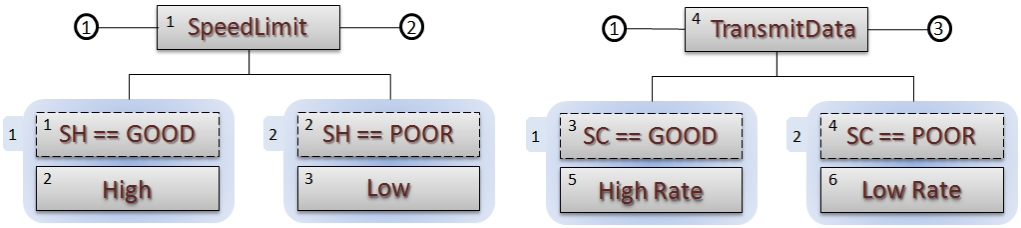


Figure 2.6: Goal trees for SpeedLimit and TransmitData goals

Table 2.1: Controlled Goal Constraint Types

Goal Index	Goal Name	State Variable	Constraint Type
1	SpeedLimit	None	Macro
2	High	`Position`	Rate
3	Low	`Position`	Rate
4	TransmitData	`DataTransmission`	Reset
5	High Rate	`DataTransmission`	Rate
6	Low Rate	`DataTransmission`	Rate
7	GetToPoint	`Position`	Transition
8	MaintPosition	`Position`	Maintenance

state variables. If the states of these two state variables change at any time during the execution, the tactic constraining that state fails and the other tactic is elaborated. For example, if the state of both passive state variables is `GOOD` initially, the first tactics of both goal trees are elaborated. However, if the `SystemHealth` state variable becomes `POOR` in the next time step, the first tactic of the SpeedLimit goal will simultaneously fail and the second tactic will elaborate in its place. The firing of the second time point will occur when the transition goal, GetToPoint, completes. At that time, the TransmitData and MaintPosition goals will be active along with one tactic of the TransmitData goal. The third time point fires, ending the goal network execution, after the constrained amount of execution time ($t_2 = 5$) has passed.

2.2 Linear Hybrid Automata

Hybrid systems exhibit discrete modes of execution that have different continuous behavior or control. Switching between discrete modes can be random, timed, or guarded by some state-based condition. Hybrid systems are very prevalent and have a range of uses; therefore, there are several different ways to model them [79]. Two of the most common ways depend on the more interesting control interface for the system; hybrid automata are focused on the discrete mode switching whereas other hybrid systems, sometimes called ODE models, are focused on the continuous dynamics in the discrete modes. In this work, a linear hybrid automata model is used, as the continuous dynamics are restricted to be piecewise constant first derivatives.

A linear hybrid automaton H consists of the following components [39]:

1. A finite, ordered list of controlled state variables and clock timers, $X = \{x_1, x_2, ..., x_n\}$.

2. A finite, ordered list of passive state variables, $\mathcal{D} = \{d_1, ..., d_m\}$. The set of discrete states (or discrete sets of states) of the state variable $d_i \in \mathcal{D}$ is $\Lambda_i = \{\lambda_1, ..., \lambda_{n_i}\}$.

3. A control graph, (V, E), where V is the set of control modes or locations of the system, and E is the set of control edges or transitions between the different modes of the system.

4. The set of invariants for each location, $inv(v)$, which are the conditions on the state variables, $X \cup \mathcal{D}$, that must be true in that location.

5. The set of flow conditions, $\psi_i : X \rightarrow X$, for location $v_i \in V$, which are the equations that dictate how state propagates in each location.

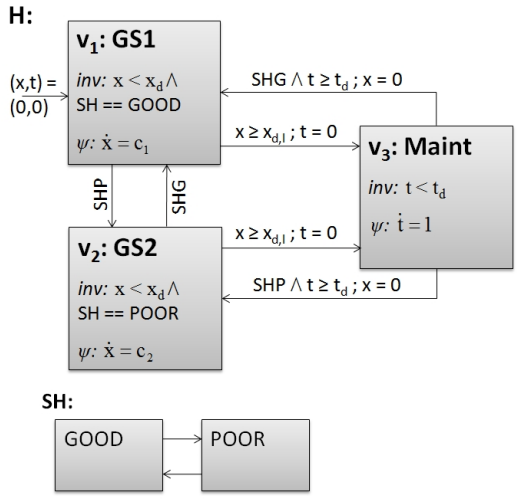


Figure 2.7: Hybrid automaton and state model example; boxes are locations or state values and arrows are edges labeled with transition conditions and resets where appropriate.

6. The set of transition conditions associated with each edge, Σ.

7. The set of transition actions or reset equations associated with each edge, A.

8. The initial conditions of the state variables, *init*.

This hybrid automaton specification can be illustrated using a simple example. Suppose there is an autonomous unmanned air vehicle (UAV) whose task is to fly to a point and then maintain that position for some amount of time. The speed at which the UAV can fly is determined by its system health (the combined health value of all its sensors). This system is described by an automaton, H and the model of the `SystemHealth` state variable, shown in Figure 2.7. The sets of state variables associated with this automaton are $X = \{x, t\}$, where x is the position and t is a timer, and $\mathcal{D} = \{\texttt{SH}\}$, where `SH` is the `SystemHealth` state variable. The locations and transition arrows (minus the conditions) compose sets V and E. The initial conditions are $(x, t, \texttt{SH}) = (0, 0, \texttt{GOOD})$. The transition conditions from v_1 and v_2 to v_3 are $x \geq x_{d,l}$ (these same transition edges have reset conditions on the timer, $t = 0$); the related invariants of v_1 and v_2 are $x < x_d$, where $0 < x_{d,l} < x_d$; $x_d, x_{d,l} \in \mathbb{R}$. This means that as soon as x reaches $x_{d,l}$, which may be the distance at which the UAV is in range of x_d, the transitions can occur, but the transition will definitely occur when $x = x_d$. Likewise, the other invariant and transition conditions dictate when the discrete transitions will occur. Finally, the differential equations with piecewise constant rates that control the flow of the variables are listed in each location.

There are several symbolic model checkers available that are capable of verifying linear hybrid

automata. These components describe a linear hybrid system that can be successfully verified using HyTech or PHAVer. The reachability analysis used in the safety verification of these hybrid automata finds the set of all states that are connected to a given initial state by a valid run. This can cause a huge explosion of the state space, however, so symbolic model checkers partition the state space into sets that are similar in the given reachability analysis. For example, given some interesting condition, PHAVer will return the set of the state space in which it is reachable; these interesting conditions given to the software are generally "unsafe" conditions for the particular system.

2.3 Stochastic Hybrid Systems

Stochastic hybrid models are hybrid systems with some uncertainty in the continuous dynamics, the discrete mode switching, or both. These systems can be classified as one of three types of models [80]. Piecewise Deterministic Markov Processes (PDMP) have random discrete transitions; the hybrid state is reset based on some probability distribution upon transitions, however between transitions, the continuous state evolution is based on ordinary differential equations. In Switching Diffusion Processes (SDP), uncertainty is included in both the continuous and discrete state evolution. A Markov chain directs the discrete switching while stochastic differential equations describe the continuous state. Finally, Stochastic Hybrid Systems (SHS) have stochastic differential equations describing the continuous state evolution while the discrete mode switching is deterministic. A discrete-time SDP will be used in this work because discrete-time execution is assumed [81].

Definition 2.3.1. A *discrete-time switching diffusion process*, $\mathcal{H}$, consists of the following components:

1. A finite, ordered list of controlled state variables and timers, $X = \{x_1, x_2, ..., x_n\}$.

2. A finite, ordered list of passive state variables, $\mathcal{D} = \{d_1, ..., d_m\}$. The set of discrete states (or discrete sets of states) of the state variable $d_i \in \mathcal{D}$ is $\Lambda_i = \{\lambda_1, ..., \lambda_{n_i}\}$.

3. The set of discrete locations, $\mathcal{V} = \{v_1, v_2, ..., v_m\}$, where $m \in \mathbb{N}$.

4. The stochastic flow of a location, $\phi : X \times \mathcal{V} \rightarrow \mathbb{R}^{d(\mathcal{V})}$, where $d(v)$ is the dimension of the continuous state space in location v.

5. The initial conditions of the system, *init*, based on some probability model.

6. The set of edges between locations, $\mathcal{E}$.

7. The transition probability associated with a given edge, $\mu : \mathcal{V} \times (X \cup \mathcal{D}) \times \mathcal{E} \rightarrow [0, 1]$, that depends on the location and the continuous and discrete state.

8. The set of transition actions or reset equations associated with each edge, A.

This definition is close to the definition of the linear hybrid automata with stochasticity added to the flow equations, and transition probabilities replacing transition conditions and invariants.

Chapter 3

Automatic Conversion Method for the Safety Verification of Goal-Based Control Systems

3.1 Introduction

The ability of goal network control programs to reconfigure as a fault response gives the control method flexibility to handle dynamic and unknown situations. However, the added complexity makes nonlinear goal network control systems difficult to check for safety with the methods used to check their linear counterparts, sequences of control commands. This constraint prohibits the use of goal networks in real applications; in systems with inherent risk, the added risk of unverified complex control systems is often not justified. Therefore, a simple and automatic way to verify goal networks is an important tool and a step towards using more fault tolerant control architectures on autonomous robots.

The main contribution of this chapter is a goal network conversion algorithm that converts goal networks into hybrid automata in a sound and complete manner; the resulting hybrid system can then be parsed into a form that is compatible with existing model checking software. In Section 3.2, more detailed information about the types of goal networks that can be converted and verified is given. In Section 3.3, a heuristic goal network conversion procedure is outlined. It is compared with the formal conversion procedure, which is described in Section 3.4; while a larger set of goal networks can be accommodated using the heuristic procedure, it is not automatic and does not come with the assurances of soundness and completeness that the formal procedure has. A description of the conversion software that is based on the formal conversion procedure is given in Section 3.5. The verification process following goal network conversion and the reverse conversion procedure

are discussed in Section 3.6. Section 3.7 concludes the chapter and summarizes the contributions.

3.2 Properties of Convertible Goal Networks

3.2.1 Structure of the Goal Network

Some restrictions exist on the types of goal networks that can be verified using the procedure described in this chapter. These restrictions mostly pertain to the structure of the goal network in time; in general, the goal networks excluded can be redesigned to fit within the restrictions.

The first restriction on convertible goal networks is that its time points must be well-ordered. This means that a scheduled goal network executes the goals in the order given. The time between time points can be constrained or unconstrained, but the order in which the time points are encountered in the goal network's execution must be set. This restriction leads to the useful group property of the goal networks, which will be defined formally in Section 3.4. In general, a group is the set of all goals that are active between two consecutive time points. Dividing the goal network in this way allows for a characterization of the discrete transitions between goals and tactics that is important for the conversion procedure.

Two other restrictions on the convertible goal networks are needed to protect the group structure. First, if a parent goal elaborates time points, all its tactics must contain the same time points. This follows from the well-ordered requirement; if a time point fires in one execution of a goal network, it must fire in all possible executions. Elaborated time points must also be respected upon re-elaboration of the parent goal; if an elaborated time point has already fired, it cannot fire again after re-elaboration.

The second requirement constraining completion goals over more than two time points. This is only possible if it can be shown that the internal time points will always fire before the completion goal is achieved. An example of an appropriate case is a robot position completion goal that elaborates an orientation, or turning, completion goal. If the position completion goal is slow enough and the turn rate fast enough, a time point at the end of the orientation goal but before the position goal's ending time point is possible.

These simple restrictions allow the goal network control program to be converted for verification. Some properties of a goal network make it easier to convert and verify, although they are not necessary for convertible goal networks to have. The first property pertains to the failure of tactics and controlled goals. The failure of tactics is assumed to be due only to the failure of the

passive goals in the tactic; the possibility of failure of a controlled goal causes a passive goal to fail instead. This is achieved by using health state variables; when the health of an actuator is poor, this is like saying that the controlled goal commanding that actuator has failed or will fail. The second property involves the transitions between tactics in a group. If the transitions are state-based (which will be formally defined later), that means that all possible states of the passively-constrained state variables in a goal network satisfy the passive constraints in some set of tactics. Goal networks with state-based transitions have nice properties that will be described in Section 3.4.

3.2.2 State Variables

The state variables constrained in a goal network can be categorized by their state models. The first type is *controllable state variables*. These state variables are directly associated with a command class in the state effects model, which means that control action is applied directly to these state variables. An example of a controllable state variable is a `HeaterSwitch` state variable that is commanded on and off to control the temperature of a device. In this model, the `Temperature` state variable is not controllable because the command is applied to the heater switch; however, in a model that instead controls the heating rate directly without using the `HeaterSwitch` state variable, the `Temperature` state variable would become a controllable state variable.

Unlike controllable state variables, *uncontrollable state variables* are not associated with any command class and also have no model dependencies on any controllable state variable. These state variables can only be passively constrained in the goal network. This designation is also dependent on the state effects model. For example, a `LADARHealth` state variable for a mobile robotic system may be dependent on the relative sun angle, which depends both on time and the position and orientation of the LADAR. This makes the `LADARHealth` state variable dependent on the `Position` state variable, which is controllable; so, the `LADARHealth` state variable would not be uncontrollable in this model. However, if the designer decides to model the relative sun angle as an independent stochastic state variable, the modeled association with the `Position` state variable disappears and the `LADARHealth` state variable becomes an uncontrollable state variable.

Dependent state variables are the last category of state variables. Dependent state variables have model dependencies on at least one controllable state variable, but do not have an associated command class. The `Temperature` state variable when there is a `HeaterSwitch` state variable is a dependent state variable, as is the `LADARHealth` state variable when the `RelativeSunAngle` state variable depends on the robot's position. Dependent state variables can be constrained by con-

trolled and passive goals. A goal on the `Temperature` state variable that elaborates constraints on the related `HeaterSwitch` state variable is an example of a controlled goal on a dependent state variable. A goal on the `LADARHealth` state variable that constrains the health to be good, but does not elaborate any control action on the `Position` state variable in order to achieve that constraint is an example of a passive goal on a dependent state variable. *Resource state variables* are a special set of dependent state variables. Resource state variables, such as power, memory, or charge cycles, are state variables that can be consumed (and in some cases, restored). Projection, which will be discussed in Section 3.3, is a useful way to handle resource state variables.

3.3 Heuristic Conversion and Verification Procedure

The goal network conversion and verification procedure can be broken up into three main parts. The conversion of the goal network to a linear hybrid automaton is the first part; the so-called goals automaton is created. The flow equations in the locations or modes of this automaton direct the propagation the controlled state variables; however, the transitions are often based on the uncontrollable and passive dependent state variables whose states must be updated in separate automata, which are created from the state models in the second part of the procedure. Finally, the system is verified against a given unsafe set using a symbolic model checker.

The heuristic version of this procedure is given in Section 3.3.2. The conversion of the goal network is formalized in the bisimulation introduced in Section 3.4 and much is automated in the software described in Section 3.5. The description of the heuristic procedure proceeds the formal procedure description to give a textual overview to the overall conversion process; the first several steps of the heuristic goals automaton procedure couple as the set-up needed to run the automatic conversion software, whose execution is described loosely by the remaining steps of the procedure. There are some additional capabilities that exist in the heuristic procedure, which are discussed in Section 3.3.4; some capabilities of MDS, such as projection, are not currently implemented in either procedure because of some severely limiting results. However, projection and how it would fit with this verification procedure are discussed in Section 3.3.3. First, some useful definitions are given.

3.3.1 Goal Network Definitions

Let $\mathcal{G}$ be the set of all goals in a goal network, where $\mathcal{G} = G \cup U$. The set $G = \{g_1^{i_1,j_1}, g_2^{i_2,j_2}, ..., g_N^{i_N,j_N}\}$ consists of all controlled goals in the goal network, where i_n is the parent goal index and j_n is the

tactic number into which the goal is elaborated, for $n = 1, ..., N$. The set of passive goals is $U = \{u_1^{i_1,j_1}, u_2^{i_2,j_2}, ..., u_M^{i_M,j_M}\}$, where i_m is the index of the goal's parent goal (which is always a controlled goal) and j_m is the goal's tactic number for $m = 1, ...M$. Controlled and passive goals are numbered independently because this notation is useful for the implementation of the conversion and verification procedure; however, $g_n^{i_n,j_n} \in \mathcal{G}$ will represent both controlled and passive goals. Let $\mathcal{T} = \{T_1, T_2, ..., T_{K+1}\}$ be the set of time points in the goal network, where $T_1 < T_2 < ... < T_{K+1}$ for increasing time. Each goal, $g_n^{i_n,j_n} \in \mathcal{G}$, has several functions associated with it.

1. start($g_n^{i_n,j_n}$) returns the goal's starting time point.

2. end($g_n^{i_n,j_n}$) returns the goal's ending time point.

3. svc($g_n^{i_n,j_n}$) returns the state variable constrained by the goal.

4. c($g_m^{i_m,j_m}, g_n^{i_n,j_n}$) returns true if the two goals have constraints that are consistent (see Definition 3.4.1) and false otherwise.

5. cons($g_n^{i_n,j_n}$) returns the constraint (or invariant) on the state variable; cons($g_n^{i_n,j_n}$) $\in Q \times \mathbb{R}$, where

$$Q = \{=, \neq, <, >, \leq, \geq, \rightarrow\}$$

   and $\rightarrow$ indicates a transition constraint.

6. entry($g_n^{i_n,j_n}$) $\in Q \times \mathbb{R}$ returns the condition on the goal's constrained state variable that must be true when entering the goal.

7. exit($g_n^{i_n,j_n}$) $\in Q \times \mathbb{R}$ returns the condition on the goal's constrained state variable that must be true before exiting the goal.

The entry logic of a passive goal is just the goal constraint and the exit logic is always true. Since passive goals constrain the conditions in which a tactic may be executed, if those conditions become false, the tactic fails. The following two functions give elaboration logic and failure destination of each tactic, which is a group of goals with the same parent index numbers (i_n) and tactic numbers (j_n).

1. startsin(i_n, j_n) returns the condition in which the tactic is elaborated initially. This is generally just a list of passive goal constraints in that tactic, though other conditions (such as tactic elaboration order) can be included.

2. failto(i_n, j_n) returns the index of the goal to which execution goes upon failure. If the goal fails to a general safing state that may be present in the control system design, the output of this function is "Safe."

The following goal classifications are important in describing the conversion of a goal network to a hybrid automaton.

Definition 3.3.1. A goal is *root goal* if it has no parent goal. It is signified by $i_n = 0$ and $j_n = 0$. Root goals are the ancestors of all other goals in a goal network.

Definition 3.3.2. Two goals $g_m^{i_m,j_m}$ and $g_n^{i_n,j_n}$ are *siblings* if $i_m = i_n \wedge j_m = j_n$. Sibling goals are goals that are elaborated from a parent goal into the same tactic; sibling goals are always compatible and always executed at the same time if active during the same time points.

Definition 3.3.3. A *completion goal* is a controlled goal with a transition constraint type; the transition constraint type requires a state variable to reach a certain value, $p \in \mathbb{R}$. The constraint is written $(\rightarrow, p)$ and the non-trivial exit condition for the goal is $(=, p)$.

3.3.2 Procedure Description

3.3.2.1 Goals Automaton

The first hybrid automaton that is created from the goal network is based on the goals; the controlled state variables are controlled by this automaton. The hybrid automata created from the uncontrollable and dependent state variables will be discussed in Section 3.3.2.2. The process to create the goals hybrid automaton is as follows. This process will be formalized in Section 3.4 and a simple example of the conversion procedure will be given there as well.

1. *State Variable Labels:* Label each state variable in the goal network as controllable, uncontrollable, or dependent.

2. *Merge Logic:* For each state variable constrained by a controlled goal, $x_i \in X$, where $X = \{\text{svc}(g_n^{i_n,j_n}) | g_n^{i_n,j_n} \in G\}$, create a merge logic table from the types of constraints imposed on the state variable in the goal network. Examples of constraint types on a mobile robot's `Position` state variable could be transition (move to a point), rate (speed limits), or a combination of these. An example of the merge logic table for this state variable is given in Table 3.1. For each pair of constraints, the conditions that allow the row constraint (r) to

Table 3.1: Example of a merge logic table for a mobile robot's `Position` state variable

		Transition	**Velocity**	**Combo**
Transition	Condition Constraint Type	$r[1] == c[1]$ $\{r[1]\}$ Transition	True $\{r[1], c[1]\}$ Combo	$r[1] == c[1]$ $\{r[1], c[2]\}$ Combo
Rate	Condition Constraint Type		True $\{\min(r[1], c[1])\}$ Rate	True $\{c[1], \min(r[1], c[2])\}$ Combo
Combo	Condition Constraint Type			$r[1] == c[1]$ $\{r[1], \min(r[2], c[2])\}$ Combo

Table 3.2: Example of a constraint properties table for a mobile robot's `Position` state variable

	Entry	**Exit**	**Dyn. Eq.**	**Reset**
Transition	True	$x >= 0.99 * r[1]$	$\dot{x} = r[1] - x$	None
Velocity	$\dot{x} \leq r[1]$	True	$\dot{x} \leq r[1]$	None
Combo	$\dot{x} \leq r[2]$	$x >= 0.99 * r[1]$	$\dot{x} = \min(r[2], (r[1] - x))$	None

merge with the column constraint (c) are given in the top field. The middle field assigns the constraint vector for the new constraint from the row and column constraints (if the conditions are met) and the bottom field gives the new constraint type.

3. *Constraint Properties:* For each state variable $x_i \in X$, create a constraint properties table. The control characteristics of each possible constraint type on the state variable are listed here. The characteristics include entry and exit logic for the goal (the conditions that must be true before the goal is entered or exited), the dynamical update equation associated with the constraint, and any control resets associated with the constraint. An example constraint properties table for the robot's `Position` state variable is given in Table 3.2, where r represents the constraint vector for each constraint type.

4. *Elaboration Logic:* For any controlled goal $g_n^{i_n,j_n} \in G$ that is a parent of another controlled goal, create an elaboration logic table if and only if the transitions between its tactics are not state-based. The elaboration logic table includes the invariant of each tactic, which are the conditions that must always be true when executing the tactic; the StartsIn logic, which are the conditions that must be true for the tactic to be initially elaborated; the failure logic,

Table 3.3: Outline of an elaboration logic table

$g_n^{i_n,j_n}$	**Invariant**	**StartsIn**	**Fail Conditions**	**Destination**
1				
2				
:				

which are conditions that cause the re-elaboration of the goal; and the corresponding failure location, whether it is another tactic or Safing. An outline of the elaboration logic table is Table 3.3. In state-based goal elaborations, the invariants of the tactics are the passive goals in each tactic, and the starts in logic, the failure conditions, and destinations are all based on these invariants.

5. *Groups:* Number each time point that is associated with a controlled goal sequentially as $\{T_1, T_2, ..., T_{K+1}\}$, where $K+1$ is the number of time points. Group goals between consecutive time points into sets $\mathcal{G}_k$, where $k = 1, 2, ..., K$. In the hybrid automaton, consecutive groups will have connectors, depicted as small empty circles, between them.

6. *Location Creation:* For each group, $\mathcal{G}_k$, $k = 1, ..., K$, find all sets of goals in the group that can be executed concurrently. These executable sets of goals must follow several rules that govern the execution of goal networks:

   (a) Goals can only execute between their constrained time points.

   (b) If a goal is executing, so must be its

      i. parent,

      ii. siblings, and

      iii. at least one of its children,

      if these goals exist.

   (c) If a root goal has elaborated goals in a group, at least one of those goals must be executing at all times. All root goals in a group execute at all times during the group's execution.

   (d) Goals in different tactics from the same parent goal cannot execute at the same time.

   Each of these sets of concurrently executable goals becomes a location.

7. *Merge Constraints:* For each location in each group, merge controlled goals constraining the same state variable. The merge logic tables give the conditions for the merge as well as the resulting constraint on the state variable. If there are more than two constraints on a state variable, merge goals pairwise until only one constraint on the state variable remains. If the merge is not possible for any pair of state constraints, the location is removed due to constraint inconsistency.

8. *State Variable Updating:* For each location in each group, use the constraints on each state variable to find the dynamic equations that describe the control and evolution of the controllable state variables in the location. If there is a time constraint on the group, add an equation for the propagation of a counter state variable.

9. *Extra Locations:* Add Success and Safing locations to the hybrid automaton.

10. *Initial Entry Transitions:* For each location in each group, create an entry transition from the preceding group connector (or initially for $\mathcal{G}_1$). The condition on this transition will be a combination of the StartsIn logic for each tactic represented in the location. The StartsIn logic can be found in the parent goal's elaboration logic table, or for parent goals that have state-based transitions, the StartsIn logic is that tactic's passive goal constraints. The logic from each tactic should be combined using a logical conjunction; eliminate any transitions whose condition logically reduces to "False."

11. *Failure Transitions:* For each location in each group, create failure transitions to other locations in the group or to Safing, if appropriate. The conditions on these transitions and their destinations depend on the failure logic of each tactic represented in the location. For goal networks with state-based transitions, the failure conditions are all possible ways the passive constraints (or invariant) of the location can be violated. The destinations of the transitions with these conditions are found by comparing the failure condition with the invariants of other locations. The invariant that is satisfied by the failure condition and is otherwise the most closely related to the original location's invariant is the destination. For other tactics, the failure conditions and destinations can be found in their parent goals' elaboration logic tables. Failure conditions of transitions with the same destination can be combined using a logical disjunction.

12. *Entry Logic and Resets:* Append the appropriate entry logic corresponding to each constraint

in each location to every initial entry transition to that location. If a location has constraints with corresponding reset equations, add resets to all incoming transitions of that location. If a group has a time constraint on its bounding time points, add a nullifying reset on the counter state variable to all initial transitions into the group.

13. *Nominal Exit Transitions:* For each location in each group, create exit transitions to the following group connector (or to the Success location for $\mathcal{G}_K$). For a location in group $\mathcal{G}_k$, the condition of this transition is either the time constraint on the bounding time points or the exit conditions of all completion constraints in the location that have T_{k+1} as their ending time points. If there are no applicable completion goals in a location, the exit condition in the absence of a time constraint is "True."

14. *Location Removal:* Remove any location that is not entered by any transitions and remove all transitions that originate at that location. Remove any other location that the goal network execution cannot reach.

15. *Initial Conditions:* Assign an initial location for the automaton and initial conditions for each of the controlled state variables.

3.3.2.2 Uncontrollable and Dependent State Variables

The process in the previous section results in a hybrid automaton for the goal network that updates the controllable state variables; the process for creating hybrid automata to update the uncontrollable and dependent state variables is described in this section. Since the transitions between discrete or continuous states for these state variables are not directly controllable, they generally happen randomly, at a given rate, or when discrete events occur. This information will be used to create the hybrid automata for these state variables and for setting up the verification problem.

The process outlined below generally describes the creation of the other hybrid automata.

1. Discretize states or rates of change of each uncontrollable or dependent state variable. Make these discrete states into locations for that state variable's automaton.

2. Using the model of the state variable, assign the appropriate dynamical equations, resets, and/or transitions to each location.

3. Assign an initial condition and location for each state variable.

3.3.2.3 Hybrid System Verification

Once all of the hybrid automata are created, the system is ready for verification. The process now becomes dependent on which software will be used to verify the system. The system will be verified against sets of incorrect or unsafe states as determined by the designer. The automata created above need to be translated from their general form into the syntax of a model checker, the unsafe set must be added, and then the system can be verified. If any changes to the hybrid automata are necessary in order to verify the system versus the unsafe set, those changes must be translated back into the original goal network. This process will be described more in depth in Section 3.6.

3.3.3 Projection

Many autonomous robotic control problems involve planning activities around a goal of maintaining a certain amount of some resource state variable, such as power or memory. The act of replanning or rescheduling based on estimates of the remaining amount of a resource state variable is part of projection in MDS. By assuming that the goal network is already scheduled in the verification problem, projection in its truest form is precluded from being verified in this way. However, there is a conservative way to verify that a goal network as scheduled will respect the resource goal constraints, especially if the choice of tactics of certain goals is based on the projected need of the given resource.

In order to include projection-induced failure and re-elaboration, extra information is necessary in the hybrid automaton. For each location, the estimated amount of resource needed in that location and the minimum and maximum forward amount of resource needed (FAN) must be calculated. The minimum and maximum FAN are the sum of the respective minimum and maximum amount of resource needed over all locations in each subsequent group. These numbers are used for the entry logic conditions that compare the actual amount of resource needed to the actual amount available. In general, if there is more resource available than is needed, the location can be entered.

The procedures for converting projection to a verifiable form are complex and have several limitations, conservatism being just one. The idea of projection in a pre-scheduled goal network severely limits its usefulness. Allowing it to be a part of the verifiable hybrid automaton puts more structure on the type of goal networks that are convertible. Therefore, it has been purposefully left out of the procedure to convert the goal networks because it was determined that the implementation makes the procedure more rigid for very little benefit.

3.3.4 Comparison with Formal Method

The heuristic conversion method described briefly in Section 3.3.2 is more flexible than the bisimulation method that will be described in Section 3.4, but at a price. The extra capabilities afforded the conversion currently disallow soundness and completeness properties and in some cases, they also prohibit the ability to automate the procedure. Without the automation of the conversion, large goal networks cannot be efficiently verified. A human converting the goal network is bound to make errors and omissions, which can adversely affect the verification efforts. Also, though there may be a way to prove the soundness and completeness of the techniques to convert a broader set of goal networks, these are not currently in place, which causes the verification to lose its value.

Currently, the automatic conversion procedure cannot handle completion goals split by a time point, though this may be easy to insert in the proof of the bisimulation. The other main difference is the restriction of the software method to systems with state-based failure transitions. This restriction is due to the proof of the bisimulation; systems with failure transitions based on order or other design choices can be converted by choosing a slightly different conversion algorithm. However, systems with state-based transitions have nice properties which will be discussed later and they are often the most robustly designed systems. Imposing arbitrary order or structure on tactics often causes unexpected trouble in the goal network's execution.

3.4 Conversion and Verification Procedure

3.4.1 Formal Description of Goal Network Executions

A valid execution of the goal network consists of a sequence of alternating flow and transition conditions,

$$\phi_{\eta_f}(t_f)...\phi_{\eta_2}(t_2)\rho_{\eta_2\eta_1}\phi_{\eta_1}(t_1)X_0, \tag{3.1}$$

where X_0 is the set of initial conditions of the controlled state variables, $\phi_{\eta_n}(t_n)$ is the set of flow conditions associated with the executable set of goals θ_{η_n} (defined below) and propagated forward in time t_n steps, and $\rho_{\eta_{n+1}\eta_n}$ is the transition between the executable sets of goals θ_{η_n} and $\theta_{\eta_{n+1}}$.

Due to the structure imposed on the time points, goals can be placed into K groups, $\mathcal{G}_k$, $k = 1, ..., K$ where

$$\mathcal{G}_k = \{g_n^{i_n,j_n} \in \mathcal{G} | \text{start}(g_n^{i_n,j_n}) \leq T_k \wedge \text{end}(g_n^{i_n,j_n}) \geq T_{k+1}\}. \tag{3.2}$$

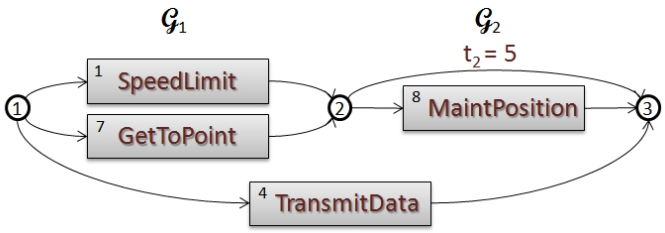


Figure 3.1: Goal network with two groups

Time points also have a constraint function, cons(T_k, T_{k+l}), that returns the time constraint between the two time points if it exists, and returns true if they are unconstrained. The amount of execution time that passes between two time points, T_k and T_{k+1}, is t_k. A simple example of a goal network with three time points, $\mathcal{T} = \{T_1, T_2, T_3\}$ and two groups, $\mathcal{G}_1$ and $\mathcal{G}_2$, is shown in Figure 3.1. The first group, $\mathcal{G}_1$, contains a completion goal; therefore the two bounding time points have no time constraint, cons$(T_1, T_2) =$ `True`. Time point T_2 fires once the completion goal has been achieved. The second group, $\mathcal{G}_2$, contains an maintenance goal and the two bounding time points have a time constraint, cons$(T_2, T_3) = [t_2 == 5]$. Time point T_3 fires when the specific amount of execution time has passed. One additional note is that the TransmitData goal is a part of both groups, $\mathcal{G}_1$ and $\mathcal{G}_2$.

The following goal relationships are important to the description of goal network executions.

Definition 3.4.1. Two goals $g_m^{i_m,j_m}$ and $g_n^{i_n,j_n}$ are *consistent* if they constrain different state variables, $svc(g_m^{i_m,j_m}) \neq svc(g_n^{i_n,j_n})$, or if $svc(g_m^{i_m,j_m}) = svc(g_n^{i_n,j_n})$ and the goals' constraints are able to be executed concurrently or merged according to the state variable's merge logic table, e.g., Table 3.1.

Definition 3.4.2. Two goals $g_m^{i_m,j_m}$ and $g_n^{i_n,j_n}$ are *compatible* if

$$\text{comp}(g_m^{i_m,j_m}, g_n^{i_n,j_n}) :=$$
$$[i_m == i_n \wedge j_m == j_n] \vee [i_m \neq i_n \wedge \text{comp}(g_{i_m}^{i_{i_m},j_{i_m}}, g_{i_n}^{i_{i_n},j_{i_n}})] \vee i_m == 0 \vee i_n == 0 \quad (3.3)$$

is true. In other words, the goals are compatible if they are in the same tactic, or if they have different parent goals and the parent goals are compatible. Root goals are compatible with all other goals by definition. Incompatible goals can never be executed at the same time in a goal network.

Certain subsets of the goals in each set $\mathcal{G}_k$ can be executed at the same time; these subsets are called executable sets, θ_η. The set of all executable sets that are built from the goals in $\mathcal{G}_k$ is Θ_k.

Definition 3.4.3. An *executable set* of goals $\theta_\eta \in \Theta_k$ is any set of goals that satisfies the following properties:

1. All goals in the executable set are active between the appropriate time points; for all $g_n^{i_n,j_n} \in \theta_\eta$, $g_n^{i_n,j_n} \in \mathcal{G}_k$.

2. All root goals in the group are in each executable set; for all $g_n^{0,0} \in \mathcal{G}_k$, $g_n^{0,0} \in \theta_\eta$.

3. If a parent goal in the executable set has child goals in the group, at least one of those child goals will also be in the executable set; for all $g_n^{i_n,j_n} \in \theta_\eta$, if there exists $g_m^{i_m,j_m} \in \mathcal{G}_k, m \neq n$, such that $i_m = n$, then there exists $g_l^{i_l,j_l} \in \theta_\eta$ such that $i_l = n$.

4. The parent goals of all goals in an executable set are also in the set; for all $g_n^{i_n,j_n} \in \theta_\eta$, if there exists $g_m^{i_m,j_m} \in \mathcal{G}_k, m \neq n$, such that $i_n = m$, then $g_m^{i_m,j_m} \in \theta_\eta$.

5. The siblings of all goals in the executable set are also in the set; for all $g_n^{i_n,j_n} \in \theta_\eta$, if there exists $g_m^{i_m,j_m} \in \mathcal{G}_k, m \neq n$, such that $i_m = i_n \wedge j_m = j_n$, then $g_m^{i_m,j_m} \in \theta_\eta$.

6. Let $\mathcal{S}_n$ be the set of goals descended from some root goal, $g_n^{0,0} \notin \mathcal{G}_k$. Then, if any of the root goal's descendants is in the group, $\mathcal{S}_n \cap \mathcal{G}_k \neq \emptyset$, at least one of those descendants is in each executable set; there exists $g_l^{i_l,j_l} \in \mathcal{S}_n \cap \mathcal{G}_k$ such that $g_l^{i_l,j_l} \in \theta_\eta$.

7. For all $g_n^{i_n,j_n}, g_m^{i_m,j_m} \in \theta_\eta$, $g_n^{i_n,j_n}$ and $g_m^{i_m,j_m}$ are compatible.

8. For all $g_n^{i_n,j_n}, g_m^{i_m,j_m} \in \theta_\eta$, $g_n^{i_n,j_n}$ and $g_m^{i_m,j_m}$ are consistent.

There are two different types of transitions between the goals in the goal network, transitions based on goal completion, $\rho^c_{\eta\mu,k} \in S^{\text{comp}}$, and transitions based on goal failure, $\rho^f_{\eta\mu,k} \in S^{\text{fail}}$, where S^{comp} and S^{fail} are the sets of all completion and failure transitions, respectively. The transition $\rho^c_{\eta\mu,k}$ is from $\theta_\eta \in \Theta_k$ to $\theta_\mu \in \Theta_{k+1}$, and

$$\rho^c_{\eta\mu,k} := \bigwedge_{\theta_\eta} \text{exit}(g_n^{i_n,j_n}) \wedge \bigwedge_{\theta_\mu} \text{entry}(g_m^{i_m,j_m}) \wedge \bigwedge_{\theta_\mu} \text{startsin}(g_m^{i_m,j_m}) \wedge \text{cons}(T_k, T_{k+1}). \tag{3.4}$$

The transition $\rho^f_{\eta\mu,k}$ is from $\theta_\eta \in \Theta_k$ to $\theta_\mu \in \Theta_k$. Let F be the set of failing passive goals in executable set θ_η and let $J = \{\forall u_n^{i_n,j_n} \in F | \text{failto}(i_n, j_n)\}$. Let $\nu_\eta \subset \theta_\eta$ be the set of all passive

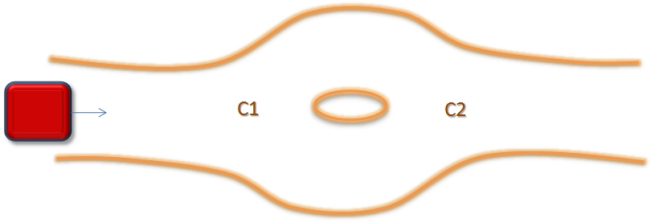


Figure 3.2: Path for the simple rover example

goals in θ_η. Then, if Safe $\in J$,

$$\rho^f_{\eta\text{Safe},k} := \bigwedge_F \neg\text{cons}(u_n^{i_n,j_n}) \wedge \bigwedge_{\nu_\eta \setminus F} \text{cons}(u_m^{i_m,j_m}). \tag{3.5}$$

Otherwise, if Safe $\notin J$,

$$\rho^f_{\eta\mu,k} := \bigwedge_F \neg\text{cons}(u_n^{i_n,j_n}) \wedge \bigwedge_{\nu_\eta \setminus F} \text{cons}(u_m^{i_m,j_m}) \wedge \bigwedge_{\nu_\mu} \text{cons}(u_l^{i_l,j_l}). \tag{3.6}$$

Only transitions whose conditions evaluate to true are taken. Valid transitions are all those whose conditions are not invariantly false.

A simple example to illustrate these concepts involves a robot with position sensors traversing a path, shown in Figure 3.2, to get to a point of interest via one of two paths, whose selection depends on the availability of the upper path. The state variables in this problem are as follows: `RobotPosition` (x), `RobotOrientation` (θ), `UpperPathAvailability` (`UP`), and `SystemHealth` (`SH`), which depends only on the sensor health states. Figure 3.3 shows the goal network for this example and Figures 3.4 and 3.5 depict the goal trees that direct the path and speed of the rover. Table 3.4 has the function outputs for some goals; goals that are similar to one listed are not shown. The rover can traverse the first segment of the path as long as the `SystemHealth` is `FAIR` or `GOOD`. The rover then decides to go to C2 via the upper or lower paths. If the estimated value of the `UpperPathAvailability` becomes `BLOCK` at any time, the robot reverses course and uses the lower path (which is assumed to always be clear). If the `SystemHealth` at any time is `POOR`, the robot safes by stopping; this option is available in all groups. Since the goal network has state-based transitions over the `SystemHealth` and the `UpperPathAvailability` state variables, the startsin() logic for all tactics is related to the accompanying passive goals.

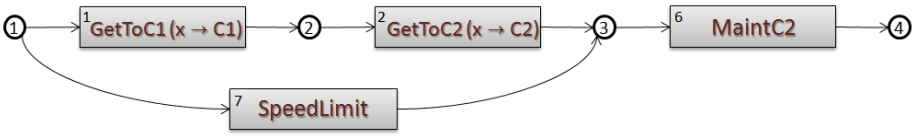


Figure 3.3: Goal network for the simple rover example

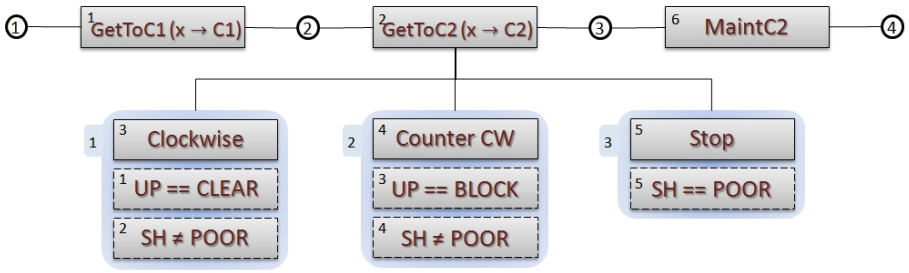


Figure 3.4: Route goal trees for the simple rover example

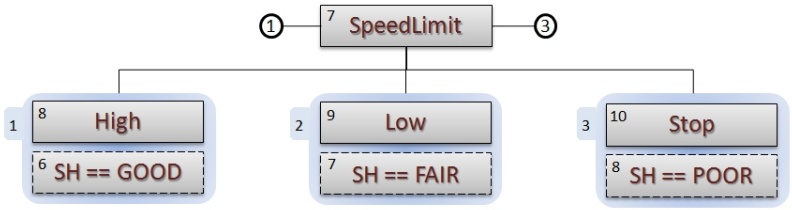


Figure 3.5: Speed limit goal tree for the simple rover example

Table 3.4: Select Goal Function Outputs for Simple Rover Example

Goal	Name	svc	cons	entry	exit
$g_1^{0,0}$	GetToC1	x	$(\rightarrow, C1)$	True	$(=, C1)$
$g_3^{2,1}$	Clockwise	θ	$(=, \mathrm{f}(x))$	True	True
$g_6^{0,0}$	MaintC2	$\dot{x}$	$(=, 0)$	$(=, 0)$	True
$g_8^{7,1}$	High	$\dot{x}$	$(\leq, v_{\text{High}})$	$(\leq, v_{\text{High}})$	True
$u_1^{2,1}$	UP == CLEAR	UP	$(=, \texttt{CLEAR})$	$(=, \texttt{CLEAR})$	True
$u_2^{2,1}$	SH $\neq$ POOR	SH	$(\neq, \texttt{POOR})$	$(\neq, \texttt{POOR})$	True

3.4.2 Procedure Description

Hybrid system analysis tools can be used to verify the safe behavior of a hybrid system; therefore, a procedure to convert goal networks into hybrid systems is an important tool for goal network verification. These goal networks can have several state variables and several layers of goal elaborations, however time points must be well-ordered, which means the time points fire in the order that they are listed in the elaboration. This restriction only states that the goal network has already been scheduled and that goals cannot switch order; it gives no restriction on the amount of time between time points. For the software, one more restriction is currently necessary; goals with non-trivial exit conditions cannot be split by a time point due to the way that the exit condition is handled.

Like the heuristic procedure, the first automaton created in the automatic conversion is called the goals automaton. This automaton has execution paths of the form

$$\psi_{\eta_f}(t_f)\tau_{\eta_f\eta_{f-1}}...\psi_{\eta_2}(t_2)\tau_{\eta_2\eta_1}\psi_{\eta_1}(t_1)X_0 \tag{3.7}$$

where X_0 is the set of initial conditions on the controlled state variables, $\psi_{\eta_n}(t_n)$ is the flow associated with location v_{η_n} for t_n time steps, and $\tau_{\eta_n\eta_{n-1}}$ is the transition from location v_{η_n} to $v_{\eta_{n-1}}$. First, some definitions are listed.

Definition 3.4.4. A *branch goal* is a controlled goal $g_n^{i_n,j_n} \in G_k$ such that for all $g_m^{i_m,j_m} \in \mathcal{G}_k, i_m \neq n$; in other words, it is a goal that has no child goals in the same group as itself. A branch goal can also be a passive goal with no controlled goal siblings, $u_n^{i_n,j_n} \in U_k$ such that for all $g_m^{i_m,j_m} \in G_k$, $i_n \neq i_m \vee j_n \neq j_m$. In a goal tree, these goals find themselves at the ends of the branches of goal elaborations.

Definition 3.4.5. Two locations, v_η and v_μ, are *compatible* if for all $g_n^{i_n,j_n} \in v_\eta$ and for all $g_m^{i_m,j_m} \in v_\mu$, $g_n^{i_n,j_n}$ and $g_m^{i_m,j_m}$ are compatible.

There are four sets of procedures used to create the goals automaton. First, the locations of the goals automaton are created. For each group of goals $\mathcal{G}_k$, $k = 1, ..., K$, a group of locations V_k is created using these procedures.

Location Creation Procedures:

1. Let $V_k = \{v_{\eta_1}, v_{\eta_2}, ..., v_{\eta_B}\}$ where B is the number of branch goals in $\mathcal{G}_k$, $v_{\eta_n} = \{g_{b_n}^{i_{b_n},j_{b_n}} | g_{b_n}^{i_{b_n},j_{b_n}}$ is a branch goal$\}$.

2. For all $g_n^{i_n,j_n}, g_m^{i_m,j_m} \in \mathcal{G}_k$ such that $g_n^{i_n,j_n} \in v_\eta$, $g_m^{i_m,j_m} \in v_\mu$, and $v_\eta, v_\mu \in V_k$, if the goals are compatible, $i_m = i_n \wedge j_m = j_n \wedge n \neq m$, then combine the locations into a new location, $v_\nu = v_\eta \cup v_\mu$, and remove v_η and v_μ from V_k.

3. For all $v_\eta \in V_k$ and for all $g_m^{i_m,j_m} \in v_\eta$:

   (a) Add to each v_η the parent goals of each $g_m^{i_m,j_m} \in v_\eta$; if there exists $g_a^{i_a,j_a} \in \mathcal{G}_k$ such that $a = i_m$ then $g_a^{i_a,j_a} \in v_\eta$.

   (b) Add to each v_η the sibling goals of each $g_m^{i_m,j_m} \in v_\eta$; if there exists $g_a^{i_a,j_a} \in \mathcal{G}_k$ such that $i_a = i_m \wedge j_a = j_m \wedge a \neq m$ then $g_a^{i_a,j_a} \in v_\eta$.

   (c) Add to each v_η the root goals in $\mathcal{G}_k$; if there exists $g_a^{0,0} \in \mathcal{G}_k$ then $g_a^{0,0} \in v_\eta$. This step is not needed, since all root goals will be added to the location with the previous two steps.

4. Combine compatible locations using the following procedure:

   (a) Let $V_k^i, i = 1$, be the set of original locations.

   (b) For all $v_\eta, v_\mu \in V_k^i, \eta \neq \mu$, if v_η and v_μ are compatible, let $v_\nu = v_\eta \cup v_\mu, v_\nu \in V_k^{i+1}$.

   (c) For all $v_\eta \in V_k^i$, if for all $v_\mu \in V_k^{i+1}$, $v_\eta \not\subseteq v_\mu$, add v_η to V_k^{i+1}. For all $v_\eta, v_\mu \in V_k^{i+1}, \eta \neq \mu$, if $v_\eta = v_\mu$, remove v_η, keeping v_μ.

   (d) Increment i. Repeat Steps (b)–(d) until for all $v_\eta, v_\mu \in V_k^i, m \neq n$, v_η and v_μ are incompatible.

   (e) Set $V_k = V_k^i$.

5. For all $v_\eta \in V_k$ and for all $g_m^{i_m,j_m}, g_l^{i_l,j_l} \in v_\eta$, if the goals are inconsistent, $\neg c(g_m^{i_m,j_m}, g_l^{i_l,j_l})$, remove v_η.

The first two steps of the procedure basically set up the initial locations to contain each of the branch goals and combines locations in which the branch goals are siblings. The next step adds to the locations all the ancestor goals and siblings goals up the goal tree from the branch goal. The parent goals may be represented in many locations, but each branch goal is represented in only one location and the combination of the goals in each location after this step is the set of all goals in the group. These first three steps guarantee that each goal in a group is represented in at least one location. From the definition of executable sets, Definition 3.4.3, properties 1, 2, 4, and 5 are

satisfied by these three steps; all goals are in the same group (1), and all root goals (2), parent goals (4), and sibling goals (5) of each goal in the location are also in each location.

The fourth step is the location combining procedure. The compatible locations are combined in such a way so that all possible combinations of compatible locations are created and so that the final outcome is a set of incompatible locations. It can be shown that this combination procedure produces all possible executable sets for each group. This step satisfies the properties 3, 6, and 7 of executable sets; Lemma 3.4.7 shows that each parent in a location has at least one child goal in the location as a result of this procedure (3). Descendants of root goals that are not in the group are present in each location because they are compatible with the root goals (and descendants) that are in the group (6), and all goals in the location are compatible (7). Finally, the last step removes locations that have inconsistent goals. This satisfies property 8 of executable sets.

Transitions for the goals automaton are created using three different procedures, one for each type of transition condition. The first two are based on the completion transitions in the goal network. First, the procedure for creating entry transitions from the preceding group connector (or initially for the first group, V_1) to each location is as follows, for all $V_k, k = 1, ..., K$.

Entry Transition Creation Procedures:

1. For all $v_\eta \in V_k$, transition edges, $e^s_{\eta,k}$, are created from the preceding group connector (or initially for $k = 1$) to the location.

2. Transition conditions for each edge are created,

$$\tau^s_{\eta,k} := \bigwedge_{v_\eta} \text{entry}(g_m^{i_m,j_m}) \wedge \bigwedge_{v_\eta} \text{startsin}(i_m, j_m), \tag{3.8}$$

   where $\tau^s_{\eta,k} \in \Sigma^s_k$.

3. For all $g_m^{i_m,j_m} \in v_\eta$, if svc($g_m^{i_m,j_m}$) returns a discrete controllable state variable, cons($g_m^{i_m,j_m}$) is added to $e^s_{\eta,k}$ as a reset action.

4. If there is a time constraint on the group, a reset setting the timer variable to zero is added to $e^s_{\eta,k}$.

5. If $\tau^s_{\eta,k}$ is invariantly false, the corresponding transition edge, $e^s_{\eta,k}$, is deleted.

The procedure for creating exit transitions from the locations to the following group connector (or to the Success location for $k = K$) is as follows, for all $V_k, k = 1, ..., K$.

Exit Transition Creation Procedures:

1. For all $v_\eta \in V_k$, the transition edges, $e^e_{\eta,k}$, are created from the location to the following group connector (or to the Success location if $k = K$).

2. Transition conditions for each edge are created,

$$\tau^e_{\eta,k} := \bigwedge_{v_\eta} \text{exit}(g^{i_m,j_m}_m), \tag{3.9}$$

   where $\tau^e_{\eta,k} \in \Sigma^e_k$.

3. Time constraints are added, if necessary, to the exit condition,

$$\tau^e_{\eta,k} := \tau^e_{\eta,k} \wedge \text{cons}(T_k, T_{k+1}). \tag{3.10}$$

4. If $\tau^e_{\eta,k}$ is invariantly false, the corresponding transition edge, $e^e_{\eta,k}$, is deleted.

Finally, the procedure for creating failure transitions between locations within groups is as follows, for all $V_k, k = 1, ..., K$.

Failure Transition Creation Procedures:

1. For all $v_\eta \in V_k$, let $\Omega_\eta = \{F_1, F_2, ...\}$ where F_m are sets that contain all possible combinations of $\alpha_n \in \mathcal{A}_\eta$, where $\mathcal{A}_\eta = \{\alpha_1, \alpha_2, ..., \alpha_N\}$. Each $\alpha_n = \text{cons}(u^{i_l,j_l}_l)$, for all $u^{i_l,j_l}_l \in v_\eta$, and each $\alpha_n \in \mathcal{A}_\eta$ is unique (for all $\alpha_n, \alpha_m \in \mathcal{A}_\eta$, $\alpha_n \neq \alpha_m$).

2. For each $F_m \in \Omega_\eta$, let the set of all failure destinations be $f_m = \{\text{failto}(i_l, j_l) | \text{cons}(u^{i_l,j_l}_l) = \alpha_n, \forall u^{i_l,j_l}_l \in v_\eta, \forall \alpha_n \in F_m\}$.

3. For all $v_\eta \in V_k$ and for all $F_m \in \Omega_\eta$, if Safe $\in f_m$, create a transition edge, $e^f_{\eta\text{Safe},k}$ from v_η to the Safing location. The transition condition associated with the edge is

$$\tau^f_{\eta\text{Safe},k} := \bigwedge_{F_m} \neg\alpha_n \wedge \bigwedge_{\mathcal{A}_\eta \setminus F_m} \alpha_n, \tag{3.11}$$

   where $\tau^f_{\eta\text{Safe},k} \in \Sigma^f_k$.

4. For all $v_\eta \in V_k$ and for all $F_m \in \Omega_n$, if Safe $\notin f_m$,

$$\tau^f_{\eta\mu,k} := \bigwedge_{F_m} \neg\alpha_n \wedge \bigwedge_{\mathcal{A}_\eta \setminus F_m} \alpha_n \wedge \bigwedge_{\mathcal{A}_\mu} \alpha_n, \tag{3.12}$$

   for some $v_\mu \in V_k$, $\mu \neq \eta$, $\tau^f_{\eta\mu,k} \in \Sigma^f_k$. If $\tau^f_{\eta\mu,k}$ is not invariantly false, create a transition edge from v_η to v_μ, $e^f_{\eta\mu,k}$, whose transition condition is $\tau^f_{\eta\mu,k}$.

5. Remove any transition edge whose condition, $\tau^f_{\eta\text{Safe},k}$ or $\tau^f_{\eta\mu,k}$ is invariantly false.

The first two steps create all possible sets of failure conditions for a given location and the set of failure destinations for each. The third and fourth steps create the transition edges and conditions, which depend on the failure destination. If the transition does not go to the Safing location, then the destination depends on which location's passive goal constraints agree with the failure conditions. Finally, the last step removes any invariantly false transition. This concludes the procedure to create the goals automaton.

Next, separate hybrid automata are created for each passively constrained state variable in the following way. These automata drive state transitions for the state variables that are not propagated by the flow equations in the goals automaton's locations.

1. The locations of the automaton for each passive state variable are created from the discrete states or discrete sets of states that are constrained in the goal network if the state variable is discrete and/or it has a non-deterministic state transition model. Otherwise, the locations are based on the different rates of change that the variable can have in its state model.

2. The transitions between the locations are based on the model of the state variable; the transitions may be modeled as non-deterministic if they are uncontrollable or dependent on something that is not modeled, such as time of day. The transition conditions are derived from the state model; therefore they may depend on state variables on which there are model dependencies.

Once the hybrid system is created, the verification work begins.

1. Specify the unsafe set. This is what the hybrid system is verified against; when the system is said to be "verified," that means that the unsafe set cannot be reached during any valid execution of the hybrid system.

2. Run the hybrid system with the unsafe set through model checking software; currently, PHAVer is the default symbolic model checking software used.

3. Make and record any modifications needed to verify the hybrid automaton. Translate these modifications into changes to the goal network.

Further explanation of the model checking and verification of the goal network will be given in the following sections.

3.4.3 Soundness and Completeness

It is possible to prove that part of the conversion procedure presented in Section 3.4.2 is a bisimulation. The goals automaton encompasses the complete set of possible executions of the goal network in its locations and transitions. By construction, the locations of the hybrid automaton correspond exactly to the executable sets of the goal network, and the transitions of the goals automaton are exactly those of the goal network. The following two lemmas show that the locations of the goals automaton correspond one to one with all of the executable sets of the goal network. The first lemma states that each executable set of goals in a group is incompatible with all others.

Lemma 3.4.6. *For all* $\Theta_k, k = 1, ..., K$ *and for all* $\theta_\eta, \theta_\mu \in \Theta_k, \eta \neq \mu$, θ_η *is incompatible with* θ_μ.

Proof. Assume two executable sets, $\theta_\eta, \theta_\mu \in \Theta_k, \eta \neq \mu$, are compatible. Let $\theta' = (\theta_\eta \cup \theta_\mu) \setminus (\theta_\eta \cap \theta_\mu)$ be the set of goals in each executable set that does not belong to the other; since $\theta_\eta \neq \theta_\mu$, then $\theta' \neq \emptyset$. For all root goals $g_n^{0,0} \in \mathcal{G}_k, g_n^{0,0} \notin \theta'$ since all root goals are a part of each executable set. Then, if there exists a child goal of the root goal $g_m^{i_m,j_m} \in \mathcal{G}_k$ such that $i_m = n$ then from condition 3 in the executable set specification in Definition 3.4.3, there exists a child of the root goal in each executable set, $g_l^{i_l,j_l} \in \theta_\eta, n = i_l$ and $g_a^{i_a,j_a} \in \theta_\mu, n = i_a$. If $l \neq a$, then the goals are in the same tactic, $j_l = j_a$, because otherwise the goals would be incompatible by definition, contradicting the assumption that θ_η and θ_μ are compatible. However, if $j_l = j_a$ then from condition 5 in the executable set specification, each goal would also belong to the other executable set, $g_l^{i_l,j_l} \in \theta_\mu$ and $g_a^{i_a,j_a} \in \theta_\eta$ because they are sibling goals, so $g_l^{i_l,j_l}, g_a^{i_a,j_a} \notin \theta'$. This logic can be applied to the children of $g_l^{i_l,j_l}$ and $g_a^{i_a,j_a}$, down to the branch goals; therefore, all descendants of the root goals in compatible locations are in both locations. So, there must exist some $g_n^{i_n,j_n} \in \theta'$ that is descended from a root goal $g_l^{0,0} \notin G_k$; let $g_n^{i_n,j_n} \in \theta_\eta$. From condition 6 in the executable set specification in Definition 3.4.3, an active descendant from $g_l^{0,0}$ must be in set θ_μ; let $g_m^{i_m,j_m} \in \theta_\mu$

also be descended from $g_l^{0,0}$, $i_m = l \wedge m \neq n$. Since $i_m = i_n$ and because the locations are compatible, then by definition $j_m = j_n$. So, from condition 5, the goals are siblings and $g_n^{i_n,j_n} \in \theta_\mu$ and $g_n^{i_n,j_n} \notin \theta'$. Therefore, $\theta' = \emptyset$, so $\theta_\eta = \theta_\mu$ and the initial assumption is false. □

Lemma 3.4.7. *Let there exist $v_\eta \in V_k$ such that there exists a goal $g_n^{i_n,j_n} \in v_\eta$ that has a descendant in the group, $g_m^{i_m,j_m} \in G_k, i_m = n$, but no descendants in the location, for all $g_l^{i_l,j_l} \in v_\eta, i_l \neq n$. Then, there exists $v_\mu \in V_k, \mu \neq \eta$ such that v_η is compatible with v_μ.*

Proof. Let V_k^1 be the set of original locations and let $v_\eta \in V_k^1$. Let $g_n^{i_n,j_n} \in v_\eta$ but let none of its children be in the same location, $g_m^{i_m,j_m} \in v_\eta, i_m \neq n$; however, the goal does have at least one descendant in the group, there exists $g_l^{i_l,j_l} \in \mathcal{G}_k$ such that $i_l = n$. Then, there exists some location $v_\mu \in V_k^1$ such that $g_l^{i_l,j_l} \in v_\mu$ because either $g_l^{i_l,j_l}$ is a branch goal, a parent goal and thus an ancestor of a branch goal, or a sibling of one of these by definition. By the conversion procedure, all branch goals, their ancestors, and all sibling goals are present in at least one initial location. By step 3 of the location creation procedure, location v_η must contain all ancestors and siblings of $g_n^{i_n,j_n}$. Since $g_n^{i_n,j_n}$ is not a branch goal (because of the existence of the child goal $g_l^{i_l,j_l}$), $g_n^{i_n,j_n}$ must be a sibling of either a branch goal or an ancestor of the branch goal present in this location due to the first two steps of the location creation procedure. Likewise, since location v_μ contains $g_l^{i_l,j_l}$, it must also contain $g_n^{i_n,j_n}$ and all its siblings and ancestors by step 3 of the location creation procedure. If $g_l^{i_l,j_l}$ is not a branch goal, it is either an ancestor or sibling of the branch goal in the location. It can be shown that these locations, v_η and v_μ, are compatible. First, both locations contain $g_n^{i_n,j_n}$ and all its siblings and ancestors; these goals are compatible with each other since locations are designed to be self-compatible. Location v_μ has goal $g_l^{i_l,j_l}$ whose parent is $g_n^{i_n,j_n}$. By design, there are no goals in v_η with the same parent. Since all remaining goals in v_μ are descended from $g_n^{i_n,j_n}$ by construction, none of the remaining goals in v_μ are in v_η and none of the remaining goals in v_μ have the same parent goals as any in v_η, so they are compatible with all the goals in v_η by definition. So, the two locations are compatible and can combine.

By induction, let $v_\eta \in V_k^i, g_n^{i_n,j_n} \in v_\eta$ such that $g_n^{i_n,j_n}$ has no child goals in v_η, for all $g_m^{i_m,j_m} \in v_\eta, i_m \neq n$, and there exists a goal in the group that is descended from $g_n^{i_n,j_n}$, $g_l^{i_l,j_l} \in G_k$ such that $i_l = n$. Because of the location combination procedure in step 4 of the location creation procedure and the transitive property of compatibility, there exists a location v_μ such that comp(v_μ, v_η) and $g_l^{i_l,j_l} \in v_\mu$, as are its siblings and descendants. The locations v_η and v_μ are compatible because of the same argument as before. The goal $g_n^{i_n,j_n}$ is in both locations and the goals in v_μ that are not

in v_η are the descendants of $g_n^{i_n,j_n}$, which are compatible with all the goals in v_η since there are no other descendants of $g_n^{i_n,j_n}$ in v_η by definition. Any other goals in v_μ are compatible with all the goals in v_μ and likewise with all the goals in v_η because of the transitive property. □

Lemma 3.4.6 says that each executable set of goals is incompatible with every other executable set. This just means that each executable set of goals in a group contains at least one different tactic from a common parent goal than every other executable set in the group. The proof is by contradiction; one can show using the properties of executable sets that if two executable sets are compatible, they are the same set. Lemma 3.4.7 states that if a location in the goals automaton contains a parent goal but none of the parent goal's children, that location will be compatible with some other location in the group. Because of the construction procedure that combines the locations in a group until all are incompatible, this lemma shows that the locations satisfy property 3 in the executable set specifications in Definition 3.4.3. The proof is by induction; one can show that this lemma is true in the initial set of locations V_k^1 and then that is also true in all following sets.

The following proposition uses the lemmas to show that all executable sets are represented by locations. It is easy to see from this proposition that the flow conditions $\phi_\eta = \psi_\eta$ for corresponding executable sets and locations.

Proposition 3.4.8. *For all $\Theta_k, k = 1, ..., K$ and for all $\theta_\eta \in \Theta_k$, there exists $v_\eta \in V_k$ such that $\theta_\eta \equiv v_\eta$.*

Proof. By steps 1–3 of the location creation procedure and because the locations created from these steps are only combined and not deleted, conditions 1, 2, 4, and 5 of the executable set specification in Definition 3.4.3 are true by construction. In step 4 of the location creation procedure, since locations are combined until they are incompatible, which is justified by Lemma 3.4.6, condition 3 of the executable set specification is true by Lemma 3.4.7. Since only compatible locations are combined, condition 7 is satisfied. By construction, all non-root goals with no parent in the group, $g_m^{i_m,j_m} \in v_\eta$ such that $i_m \neq 0 \wedge g_{i_m}^{i_{i_m},j_{i_m}} \notin \mathcal{G}_k$, $g_m^{i_m,j_m}$ will appear in at least one initial location ($v_\eta \in V_k^1$) which is compatible with all initial locations containing $g_l^{0,0} \in \mathcal{G}_k$ and incompatible only with initial locations that have other goals from the same parent. Therefore, a representative goal from the set of descendants will be present in every location in the group, so, condition 6 of the executable set specifications is satisfied. Finally, step 5 assures that all goals in the locations are consistent, which satisfies condition 8 of the executable set specifications in Definition 3.4.3. Since the procedure is designed to make all compatible combinations and the locations created satisfy the

executable set specifications, all executable sets of goal are represented by exactly one location, since duplicate locations are removed (step 4c). □

The proof of Proposition 3.4.8 uses the procedure steps for the location creation and the two previous lemmas to show that all of the executable set properties are satisfied by the locations that result from the location creation procedure. Likewise, the two following lemmas relate the transitions of the goal network to the transitions of the hybrid automaton and the proofs are also by construction using the transition creation procedures.

Lemma 3.4.9. *For all $\rho^c_{\eta\mu} \in S^{comp}$, there exists an equivalent transition $\tau^c_{\eta\mu,k} \in \Sigma^c_k = \Sigma^e_k \times \Sigma^s_{k+1}$.*

Proof. By Proposition 3.4.8, each executable set $\theta_\eta \in \Theta_k$ is represented by a location $v_\eta \in V_k$; also let $\theta_\mu \in \Theta_{k+1}$ be represented by $v_\mu \in V_{k+1}$. Transitions between each $\theta_\eta \in \Theta_k$ and each $\theta_\mu \in \Theta_{k+1}$ are given by $\rho^c_{\eta\mu,k}$, defined in Eq. 3.4. In the goal network, if there exists $g^{i_m,j_m}_m \in \theta_\mu$ that constrains a discrete controllable state variable, the constraint cons(g^{i_m,j_m}_m) is executed upon entering θ_μ, so is considered to happen at the entry transition.

A transition $\tau^c_{\eta\mu,k} \in \Sigma^c_k = \Sigma^e_k \times \Sigma^s_{k+1}$ is defined to be the composition of two transitions, $\tau^e_{\eta,k} \circ \tau^s_{\mu,k+1}$. Using steps 2–3 of the exit and entry transition creation procedures, it is obvious that $\tau^c_{\eta\mu,k} = \rho^c_{\eta\mu,k}$ and the above statement is true. □

Lemma 3.4.10. *For all $\rho^f_{\eta\mu,k} \in S^{fail}$ and for all $\rho^f_{\eta Safe,k} \in S^{fail}$ in the goal network, there exists an equivalent transition $\tau^f_{\eta\mu,k} \in \Sigma^f_k$ and $\tau^f_{\eta Safe,k} \in \Sigma^f_k$, respectively, in the resulting hybrid automaton.*

Proof. By Proposition 3.4.8, for all $\theta_\eta \in \Theta_k$, there exists $v_\eta \in V_k$ which corresponds exactly. For some set $J = \{u^{i_n,j_n}_n, ...\}$ of failing passive goal conditions, $\rho^f_{\eta\mu,k}$ ($\rho^f_{\eta\text{Safe},k}$) is defined by Eq. 3.6 (3.5) if $\rho^f_{\eta\mu,k}$ ($\rho^f_{\eta\text{Safe},k}$) is not invariantly false. The transition between the corresponding locations, $v_\eta, v_\mu \in V_k$, is given by Eq. 3.12, which is equivalent to $\rho^f_{\eta\mu,k}$ given the construction of the constraints $\alpha_n \in F_m$ in the first step of the failure transition creation procedure. For the transition to Safing, the transition is given by Eq. 3.11. By Proposition 3.4.8, $\mathcal{A}_\eta = \nu_\eta$ and by construction of Ω_η, each $J \equiv F_m$ for some $F_m \in \Omega_\eta$. Therefore, $\rho^f_{\eta\mu,k} = \tau^f_{\eta\mu,k}$ and $\rho^f_{\eta\text{Safe},k} = \tau^f_{\eta\text{Safe},k}$ between corresponding locations. □

Finally, Lemma 3.4.11 proves that the transitions are the same between the goal network and the goals automaton and Theorem 3.4.12 proves that the conversion procedure is a bisimulation.

Lemma 3.4.11. *All transitions of the goal network are represented in the goals automaton.*

Proof. Since only two types of transitions are allowed in the goal network, this statement is true due to Lemmas 3.4.9 and 3.4.10. □

Theorem 3.4.12. *The conversion procedure is a bisimulation between the goal network and the goals automaton.*

Proof. By Proposition 3.4.8 and Lemma 3.4.11, all executions of the goal network are represented by paths in the hybrid automaton constructed from the goal network by using the conversion procedure. Because of this, all executions of the goal network are represented by an execution path through the hybrid automaton. There are no executions in the hybrid automaton that do not represent an execution of the goal network because the definition of executable sets, Definition 3.4.3, states that every set of goals that has the given properties is an executable set, and each location created has those properties (Proposition 3.4.8). Likewise, each transition in the hybrid automaton was constructed from a corresponding transition in the goal network, so the hybrid system is an exact representation of all the possible executions of the goal network. □

Therefore, the conversion procedure is sound in that if the hybrid automaton is verified for some unsafe set, the goal network is also verified. This is easy to see since every execution path in the goal network is represented in the hybrid automaton; so, if there exists a path in the goal network in which the given unsafe set is reachable, that path will also be present in the hybrid automaton. The conversion procedure is also complete, in that if the goal network is verifiable, the hybrid automaton will also be verifiable. There are no extra execution paths in the hybrid automaton that are not present in the goal network; in fact, there is a way to rebuild the original goal network and goal logic from the hybrid automaton, which is outlined in Section 3.6.2.

3.4.4 Simple Rover Example

The conversion and verification procedure can be illustrated using the simple rover example introduced in Section 3.4.1. The same state variables are used in this example, and both `Position` and `Orientation` are controllable state variables and the `UpperPathAvailability` and `SystemHealth` state variables are uncontrollable. The startsin() logic of the speed limit tactics are based on the accompanying passive goals. The goal network has state-based transitions and all failure transitions are based on the passive goals in each tactic. All controlled goal combinations in the goal network are consistent.

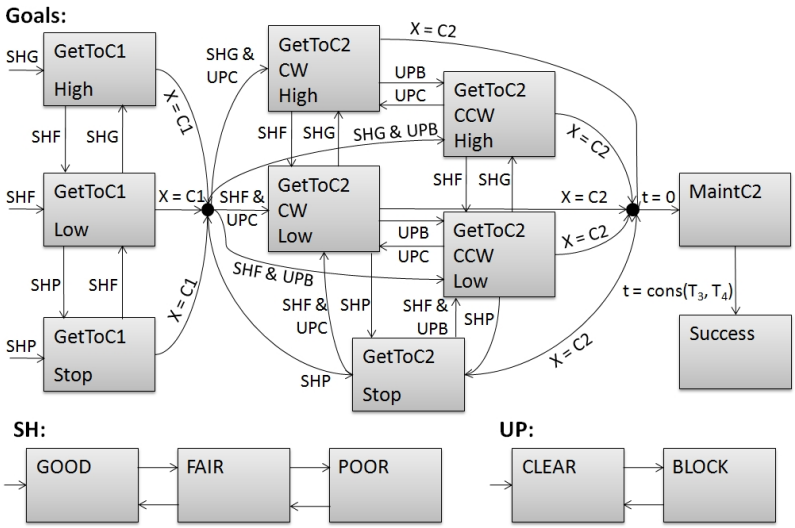


Figure 3.6: Automata for rover example

The goal network has four time points and therefore three groups, which are shown in Figure 3.6. The first group, V_1, has four sets of branch goal locations ($\{g_1^{0,0}\}$, $\{g_8^{7,1}, u_6^{7,1}\}$, $\{g_9^{7,2}, u_7^{7,2}\}$, and $\{g_{10}^{7,3}, u_8^{7,3}\}$) from steps 1 and 2 of the location creation algorithm that combine to form three incompatible locations in step 4 of the location creation algorithm, created from the combination of the GetToC1 root goal, $g_1^{0,0}$, and the three tactics of the Speed Limit root goal, $g_7^{0,0}$ ($\{g_1^{0,0}, g_7^{0,0}, g_8^{7,1}, u_6^{7,1}\}$, $\{g_1^{0,0}, g_7^{0,0}, g_9^{7,2}, u_7^{7,2}\}$, and $\{g_1^{0,0}, g_7^{0,0}, g_{10}^{7,3}, u_8^{7,3}\}$). The second group, V_2, starts with six sets of sibling branch goals that combine into a total of nine locations before consistency is checked, which covers all possible execution paths of the goal network between those time points. However, four locations were removed because of sets of inconsistent passive goals in step 5 of the location creation algorithm ($\{u_2^{2,1}, u_8^{7,3}\}$, $\{u_4^{2,2}, u_8^{7,3}\}$, $\{u_5^{2,3}, u_6^{7,1}\}$, and $\{u_5^{2,3}, u_7^{7,2}\}$). The third group, V_3, has only one goal, and therefore only one location.

The transitions into the locations either initially (V_1) or from the group connector are conditioned by startsin() elaboration logic, which is just the accompanying passive goal constraints in each tactic and entry() transition logic contributions from all goals in the group (not shown for clarity). The failure transitions between the locations in the groups are state-based; they are based on the failing and non-failing conditions of the invariant of the originating location and the invariant of the accepting location. The transition logic out of the locations to the following group connector or to the Success location (V_3) are the exit() logic conditions for each of the completion goals present in the location ($g_1^{0,0}$ and $g_2^{0,0}$). The final version of the goals automaton can be found in Figure 3.6.

The `UpperPathAvailability` and the `SystemHealth` state variables are the two un-

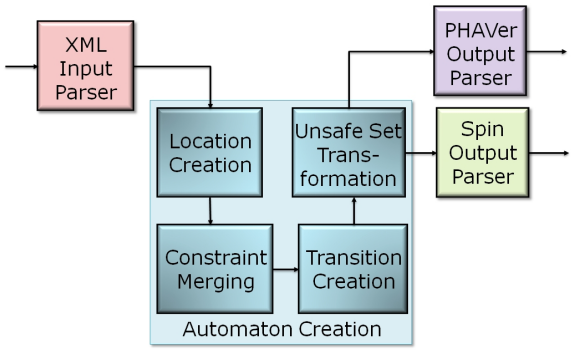


Figure 3.7: Flow chart of the conversion software execution

controllable state variables, which can be modeled as having two and three discrete state values, respectively. These state values become locations with non-deterministic transitions between them. The `SystemHealth` and `UpperPathAvailability` automata are shown in Figure 3.6. Finally, the unsafe set is determined; this is any condition that the designer decides the rover should never reach. The automata and thus the goal network can now be verified using model checking software.

3.5 Conversion Software Design

An automatic goal network conversion software that is based on the bisimulation described in the previous section takes a description of the goal network as an input and outputs a file. The output file can be input into an existing model checker for verification. The software is written in Mathematica because of the list structure it employs and its extensive library of pattern-matching functions. The software has many parts: the input parser takes an XML file with goal information, state variable models and unsafe set specifications; the automaton creation algorithm transforms the goal network information into a hybrid automaton and outputs a general form of that automaton; and the output parsers create input files to existing model checking software from the converted hybrid automaton, state variable models, and the unsafe set. The general outline for the structure of the conversion software is shown in Figure 3.7. In addition to the input and output parsers, there are four main parts to the actual conversion algorithm: location creation, constraint merging, transition creation, and unsafe set transformation. All parts of the software will be described in this section.

3.5.1 Input Parser

The conversion software's input parser takes an XML file with a specified structure and translates it into several lists that the Mathematica code can use. The input data includes several things. First, all controlled state variables are given along with all possible control modes (ways a goal could constrain the state variable). Included with the control mode information is constraint merging logic. The merge logic information for each state variable constrained by the controlled goals is directly related to the information given in the example merge logic table given in Table 3.1 for the robot's `Position` state variable. The conditions that may cause constraints to be inconsistent are given, as are the values and type of the new constraint if the original constraints can be merged. The other information included with the controlled state variables is the conditions that must be true for the goal network execution to enter or exit a given constraint type, the dynamical equation for each control mode, any reset associated with each constraint type, and the state variable's initial condition.

Next, each passive state variable is listed with its state model. These models are either non-deterministic or dependent (modeled) on other state variables. An example of a stochastic state variable could be the health of a sensor that is modeled to fail at some stochastic rate. Likewise, a sensor health state variable could have state transitions that depend on other state variables included in the model, such as a `LADARHealth` state variable that depends on the relative sun position and the amount of dust. Dependent state variables are always modeled on other state variables and often, if the state variable is continuous and constrained in both controlled and passive goals, the state variable's model will be rate-driven. This means that the discrete modes of these state variables' models have the different rates of change of the continuous state variable. Examples of this are `Temperature` or `Power` state variables whose rates of change depend on heaters or actuators being on or off.

The goals in the goal network are listed with all necessary tactic information. The time points bounding each goal, the state variable constrained, the type of constraint, and constraint value are included with each non-macro goal. For each parent goal, a list of the child goals separated into tactics is given. Controlled and passive goals are listed for each tactic and in the overall goal list; they are differentiated by several things, including the state variable constrained and the type of constraint. In some cases, the failure transitions into the Safing location are explicitly listed for each tactic, though this is not necessary.

Finally, any time constraints between time points and unsafe conditions are listed. For the unsafe set, the state variables constrained, the type of constraint on the state variable, and the constrained value are given for each unsafe condition. DTD files for the PHAVer and Spin XML input files can be found in Appendix A. The Spin version has some differences, most notably the absence of the unsafe set specification structure.

3.5.2 Automaton Creation Algorithm

The automaton creation part of the conversion software is made up of four main algorithms. Two of the four main parts of the conversion software follow the conversion procedure outlined in Section 3.4, the location creation and the transition creation algorithms. The constraint merging algorithm is important for the representation of the hybrid system in the model checking software, but is not invertible and so is not part of the bisimulation. The unsafe set transformation uses the converted hybrid automaton and the original unsafe set specification to put the unsafe set into a form that PHAVer can understand and is not run when a different model checker is used.

The location creation algorithm follows the procedure outlined in Section 3.4 to place the goals into groups and then to enumerate the locations in each group. Unlike the bisimulation conversion procedure, inconsistent locations are created in this algorithm and then are handled in the constraint merging algorithm. The locations are also assigned names based on the branch goals that are present; the names are used in the model checking software.

The constraint merging algorithm deals with both passive and controlled goal constraints. Passive goal constraints on the same state variable are inconsistent if the state value constrained is different. Controlled goal constraints are more difficult, as certain conditions may need to be met before the constraints are considered to be consistent. If the constraints (passive or controlled) are inconsistent, the location is removed. However, consistent controlled constraints are merged within each location until only one resulting constraint per state variable remains. This algorithm also assigns dynamical update equations and reset equations (if necessary) to each location once the final merged constraints have been found.

The transition creation algorithm follows the procedures outlined in Section 3.4 for creating all three types of transitions. The number of failure transitions between each location often becomes prohibitively large when all possible combinations of failure conditions are considered. So, there is an option to only find the single point failure transitions when circumstances allow. The assumption is then that either zero time can be spent in a location (multiple transitions can be taken in a single

time step to deal with multiple simultaneous failures) or that multiple failures do not cause an unsafe condition. Only the latter assumption can be made when verifying with PHAVer. A small location removal algorithm is also included in the transition creation code. The location removal algorithm checks if any location lacks entry conditions and if so, removes the location and all other failure transitions originating from that location. The algorithm also checks for other locations that would warrant removal, however it can be shown that none of these conditions will ever occur due to the way transitions and locations are created.

Finally, when PHAVer is used as the verification software, the unsafe set transformation algorithm takes the set of unsafe conditions and transforms them into a form that PHAVer can use. PHAVer cannot check rate conditions, though these may be common unsafe set specifications; an example is checking the speed of a rover when its sensor health state variables are degraded. However, this algorithm can search through the goals automaton and find all locations in which the rate conditions are satisfied. These locations are then listed with the other state variable constraints in the unsafe set specification. The goals automaton and the transformed unsafe set specification (PHAVer only) are then sent to the appropriate output file creation algorithm.

3.5.3 Output File Creation

The goals automaton and all of the passive state variables' automata are output in a very generic form so that they may be used with an output file creation algorithm that translates the lists into code for any model checker that uses automata theory to verify systems. Currently, two output file creation algorithms are available, one for PHAVer and one that outputs Promela code for the Spin model checker. The final output of these algorithms is a file that can be run through the respective model checker.

The PHAVer output file creator has some special code to create the synchronization labels that are appropriate given the unsafe set. The synchronization labels, or synclabs, are used to create a relationship between transitions in different automata. For example, a transition of a `SensorHealth` state variable from `GOOD` to `POOR` may cause a failure in the location that is executing in the goals automaton. For verification purposes, it may be important that the goals automaton executes the appropriate failure transition immediately, rather than in the next time step. Otherwise, the unsafe set may be satisfied momentarily, even though the appropriate logic is in place to ensure that safety is maintained. The file creator uses the specified unsafe set to find transitions between the goals automaton and the passive state variable automata that must be synchronized and assigns an

appropriate synclab to both transitions. Since the PHAVer output file creator deals with the unsafe set, the file created can be immediately input into PHAVer for verification with no modifications.

3.6 Goal Network Verification

3.6.1 Working with Model Checkers

Once a goal network has been converted and an appropriate model checker input file has been created, the verification work begins. The conversion algorithm is capable of handling goal networks that produce hundreds of locations and thousands of transitions; an example with over 500 locations and thousands of transitions takes less than five hours to convert. However, the model checking software often cannot verify systems this large because of the state space explosion. Therefore, some abstractions and reduction techniques are needed.

In many cases, the group structure of the convertible goal networks can be leveraged to reduce the size of the verification problem. As long as the unsafe set does not have dependencies on the completion goal(s) in a group, the groups can be verified individually. The initial condition is a concern when verifying groups other than G_1 individually, however, there is often an acceptable solution.

The state space explosion problem benefits from the reduction in the numbers of locations and automata. Oftentimes, the models of the passively constrained state variables can be adjusted in order to reduce the total number of states. Creating derived state variables from state variables that are related by some model is one way to reduce the state space. A derived state variable is a non-physical state variable whose state propagation completely depends on two or more passive state variables. The `SystemHealth` state variable is an example of a derived state variable. It is modeled from the states of several sensor health state variables. To reduce the state space, instead of modeling each sensor health state variable and including each of their automata, they can be replaced by the `SystemHealth` state variable. Removing unused states and combining states that are always constrained together are other ways to reduce the state space. Finally, discretizing continuous state variables can help reduce the complexity of the verification problem.

3.6.2 Reverse Conversion Procedure

Once the verification has been completed on the hybrid system, if any changes had to be made to the system to accomplish the verification, these changes must be translated back to the original goal network. Since the conversion procedure for certain goal networks is a bisimulation, there must be a procedure to revert a converted hybrid system back to a bisimilar goal network. Such a reverse conversion procedure has been designed, though it is very restricted in the types of hybrid automata that it can handle. There are several restrictions on the original goal network and conversions required for the reverse conversion procedure. One is that constraint merging is not part of the bisimulation and so the locations in the hybrid automaton must have each separate controlled goal constraint listed; also, each controlled constraint must be unique (or at least uniquely labeled) and each root goal in a group must directly elaborate a unique set of passive constraints. Another of these requirements is that the hybrid automaton must have state-based transitions and each elaborated tactic must have at least one controlled goal because of assumptions made in the reverse conversion procedure. The basic procedure for finding the goal network associated with a hybrid system is described here. This algorithm has been automated and can be used for the special class of goal networks that satisfy the assumptions.

Let there be locations $v_\eta \in V_k$ for each group V_k, $i = 1, ..., K$. Each location has two sets of constraints, (by abuse of notation) cons(v_η) is the set of active constraints and inv(v_η) is the set of unique passive constraints in the location. Let the set of passive constraints in V_k be

$$P_k = \bigcup_{v_\eta \in V_k} \text{inv}(v_\eta). \tag{3.13}$$

Let the set of active constraints in V_k be

$$C_k = \bigcup_{v_\eta \in V_k} \text{cons}(v_\eta). \tag{3.14}$$

The procedure is as follows:

1. Create location sets for each passive and active constraint in $p_j \in P_k$ and $c_i \in C_k$, respec-

tively.

$$\text{loc}(p_j) = \{v_\eta | v_\eta \in V_k, p_j \in \text{inv}(v_\eta)\} \tag{3.15}$$

$$\text{loc}(c_i) = \{v_\eta | v_\eta \in V_k, c_i \in \text{cons}(v_\eta)\} \tag{3.16}$$

2. Find the non-macro root goals: $R_k = \{c_i | \text{loc}(c_i) = V_k\}$. Remove these constraints from the constraint list: $C_k = C_k \setminus R_k$. Make the constraints in R_k into goals, for all $c_i \in R_k$, let $c_i = \text{cons}(g_{r_i}^{0,0})$, $g_{r_i}^{0,0} \in \mathcal{G}_k$.

3. Find the directly elaborated child goals of the root goal(s) by comparing locations sets between the passive and active constraints. The root goals' child goals are all constraints such that for any $p_j \in P_k$, $\text{loc}(p_j) = \text{loc}(c_i)$. Controlled constraints that are associated with inconsistent passive goal constraints are incompatible. If more than one controlled constraint matches the same passive constraint, those controlled constraints belong to sibling goals. Place all constraints that satisfy this condition in a list by parent goal and tactic, which can be deduced from the compatibility of the goals and the consistency of the passive goal constraints. Assign each constraint a goal index and place the goals in a set of potential parent goals, $\mathcal{P}$.

4. For each goal $g_n \in \mathcal{P}$, find its child goals, if any.

   (a) Find groups of all possible constraints, $C_{k,i}^n$, such that the disjoint location sets of the constraints cover the location set of the goal,

   $$\text{loc}(\text{cons}(g_n)) = \bigcup_{c_j \in C_{k,i}^n} \text{loc}(c_j),$$

   but for any $c_j, c_l \in C_{k,i}^n$, $\text{loc}(c_j) \cap \text{loc}(c_l) = \emptyset$.

   (b) Let $C_k^n = \{C_{k,1}^n, ..., C_{k,I}^n\}$. Find all the sets in C_k^n that have the smallest number of constraints,

   $$\bar{C}_k^n = \{C_{k,i}^n | \min_{C_{k,i}^n \in C_k^n} |C_{k,i}^n|\}. \tag{3.17}$$

   (c) Let the set of child goals be

   $$\mathcal{C}_k^n = \bigcap_{C_{k,i}^n \in \bar{C}_k^n} C_{k,i}^n. \tag{3.18}$$

Create a goal (and tactic) with a new index for all $c_j \in \mathcal{C}_k^n$, $c_j = \text{cons}(g_{m_j}^{n,j})$, and $\mathcal{G}_k = \mathcal{G}_k \cup \{g_{m_j}^{n,j}\}$. By construction, each goal in this set will be incompatible. Remove these constraints from the unplaced constraint list, $C_k = C_k \setminus \mathcal{C}_k^n$. Place all new goals in the potential parent set while removing the current parent goal, $\mathcal{P} = (\mathcal{P} \setminus \{g_n\}) \cup \{g_{m_j}^{n,j}\}$.

(d) The remaining "uncertain" constraints are grouped together in a similar way,

$$Z_k^n = \Big(\bigcup_{C_{k,i}^n \in \bar{C}_k^n} C_{k,i}^n \Big) \setminus \mathcal{C}_k^n. \tag{3.19}$$

Group the constraints in the uncertain set Z_k^n into sets of constraints that have the same location set. Then, for each subset $Z_{k,i}^n \in Z_k^n$, add $Z_{k,i}^n$ to the set of potential parents, $\mathcal{P} = \mathcal{P} \cup \{Z_{k,i}^n\}$.

Repeat this step until $\mathcal{P} = \emptyset$.

5. Identify as many of the constraints in the uncertain set as possible. Constraints that occur in only one uncertain set, $Z_{k,i}^n$, are sibling goals that belong to a new tactic of the goal, g_n, associated with the uncertain set. The placement of other uncertain constraints may be determined by comparing the state variables constrained between it and the potential parents.

6. Create macro root goals for incompatible goal sets with no parents. Assign the parent and tactic information to the goals that are lacking it.

7. Determine the starting and ending time points of each goal in G_k by comparing goals and constraints across consecutive groups.

Remark 1. If there is only one set $C_{k,i}^n \in \bar{C}_k^n$, the constraints in that set represent the children goals elaborated into different tactics of the potential parent goal. If there is more than one set, there is some uncertainty as to which goals are the children of the potential parent goal. One condition in which this uncertainty arises is when a potential parent goal elaborates a tactic with controlled sibling goals.

The output of this procedure is a goal network that may have some constraints that are unassigned. For goal networks with simple constraints, no uncertain goals should remain. The many limitations on this reverse conversion procedure indicate that there may be a better solution to this problem; however, as described later, the necessity for a procedure like this may not exist.

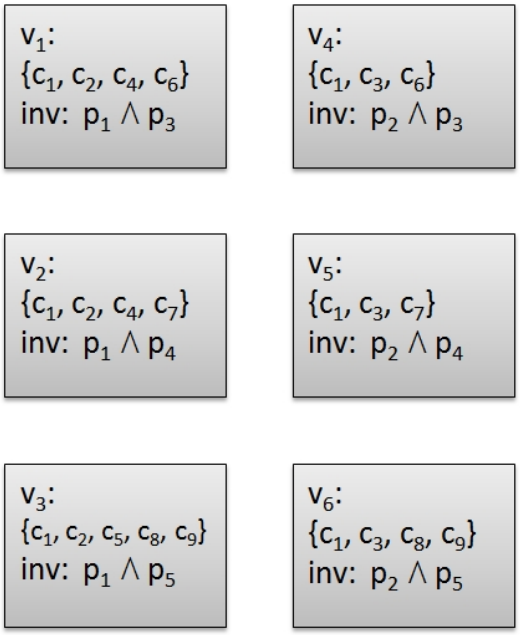


Figure 3.8: Simple hybrid system for reverse conversion example. Transitions are omitted for clarity.

A simple hybrid system example is shown in Figure 3.8. The six locations have the following numbered controlled constraints in their flow equations and resets and passive constraints in their invariants. There are two sets of incompatible passive constraints, $\{p_1, p_2\}$ and $\{p_3, p_4, p_5\}$. Table 3.5 gives the location sets for each constraint along with its status or associated passive constraint. Constraint c_1 is present in every location, and so is a root goal. Constraints c_8 and c_9 have the same location sets as each other and as passive constraint p_5, which indicates that they are sibling goals which are directly descended from a root goal. Every constraint except c_4 and c_5 are either root goals or are directly descended from a root goal, and so these two constraints make up the set $C_1 \setminus \mathcal{P}$. It is easy to see that these constraints descend from controlled constraint c_2. Finally, since only one root goal was found for two goal trees, the second root goal must be a macro goal; the root goal set is $R_1 = \{g_1^{0,0}, g_{10}^{0,0}\}$. The converted goal trees are shown in Figure 3.9.

3.7 Conclusion

The goal network conversion software presented is capable of quickly and accurately converting goal networks into a bisimilar linear hybrid automata that can be verified using existing symbolic model checking software such as PHAVer. The proofs of soundness and completeness of the con-

Table 3.5: Constraint Properties in Reverse Conversion Example

Constraint	Location Set	Goal	Associated Passive Constraint
c_1	V_1	$g_1^{0,0}$	None
c_2	$\{v_1, v_2, v_3\}$	$g_2^{r_1,1}$	p_1
c_3	$\{v_4, v_5, v_6\}$	$g_3^{r_1,2}$	p_2
c_4	$\{v_1, v_2\}$	$g_4^{2,1}$	$p_3 \vee p_4$
c_5	$\{v_3\}$	$g_5^{2,2}$	p_5
c_6	$\{v_1, v_4\}$	$g_6^{r_2,1}$	p_3
c_7	$\{v_2, v_5\}$	$g_7^{r_2,2}$	p_4
c_8	$\{v_3, v_6\}$	$g_8^{r_2,3}$	p_5
c_9	$\{v_3, v_6\}$	$g_9^{r_2,3}$	p_5
p_1	$\{v_1, v_2, v_3\}$		
p_2	$\{v_4, v_5, v_6\}$		
p_3	$\{v_1, v_4\}$		
p_4	$\{v_2, v_5\}$		
p_5	$\{v_3, v_6\}$		

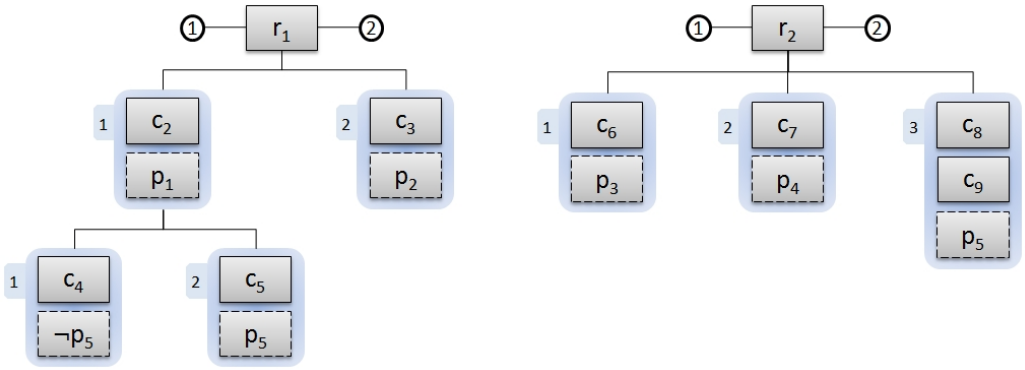


Figure 3.9: Converted goal trees for reverse conversion example

version procedure are important to validate using symbolic model checkers to verify the resulting hybrid system and applying the verification result back to the goal network. Since so much work has been done on the verification of hybrid systems, this is a useful first step towards the efficient verification of goal network control programs. However, the size and complexity of the goal networks that can be verified is subject to the constraints imposed by the symbolic model checker used; the verification method introduced in the next chapter handles much larger systems by imposing some common-sense structure on the goal network design.

Chapter 4

Efficient Verification for Systems with State-Based Transitions

4.1 Introduction

Goal network control programs can be converted to hybrid systems using the bisimulation procedure introduced in Chapter 3 and then verified using existing model checking software. However, this approach is restricted by the symbolic model checker used in the verification; oftentimes, to use these software programs to verify real systems, abstraction, model reduction, and overapproximation of the system is necessary. The limiting factor in the use of these symbolic model checkers is often the number of state variables in the system, which for real systems can be very large. The conversion procedure also has difficulty handling many passive state variables because of the way failure transitions are created. Some limiting assumptions can improve the performance of the conversion software, but at a cost.

The main contribution of this chapter is the design for verification software tool and the resulting verification algorithm. The design tool used to create goal networks with state-based transitions (defined in Section 4.2), the SBT Checker, is introduced in Section 4.3. The verification software, InVeriant, is described in Section 4.4. The application of the InVeriant model checker to a class of linear hybrid systems is discussed in Section 4.5. The capabilities, strengths and weaknesses of this verification approach are discussed in Section 4.6, followed by a summary of the contributions in Section 4.7.

4.2 State-Based Transitions

A class of goal networks, ones with state-based transitions, have special properties in the bisimilar automata.

Definition 4.2.1. Let $\mathcal{D}_k = \{d_1, d_2, ..., d_{n_k}\}$ be the set of state variables constrained by passive goals in U_k. Then, let Γ_k be the passive state space, $\Gamma_k = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_{n_k}$. If for each state $\gamma_i \in \Gamma_k$, there exists some executable set, $\theta_j \in \Theta_k$ such that the passive state satisfies the passive constraints, $\gamma_i \models \text{pcons}(\theta_j)$ for each group, $k = 1, ..., K$, and the elaboration conditions for each parent goal are based only on the states of the system, then the goal network has *state-based transitions*.

An example goal tree that does not have state-based transitions is shown in Figure 4.1. The speed limit root goal has three tactics constraining two passive state variables, `SystemHealth` and `PositionUncertainty`, whose models are included in Figure 4.1. The passive state space is also shown; it is obvious that two states (`SH` $==$ `GOOD` $\wedge$ `PU` $==$ `HIGH` and `SH` $==$ `POOR` $\wedge$ `PU` $==$ `LOW`) do not satisfy the passive constraints in any tactic. Therefore, this goal tree does not have state-based transitions. While this construction is valid, there is no way to predict what will happen to the execution when these states occur; the current tactic would fail but since there is no tactic associated with these states, any of the tactics may be chosen as the execution of the goal network breaks down. However, with two changes, the goal tree in Figure 4.2 does have state-based transitions, which is obvious from the passive state representation.

The assumption that the goal elaboration is based only on the states of the system follows the design philosophy of State Analysis and MDS. This simply says that there is no predefined order to the tactics of any goal; instead, the passive state constraints control goal elaboration. Since goal networks are bisimilar with hybrid systems, the definition of state-based transitions also applies to them. In a converted hybrid system with state-based transitions, each passive state $\gamma_i \in \Gamma_k$ satisfies the invariant, $\gamma_i \models \text{inv}(v_j)$, of some location $v_j \in V_k$ for all $k = 1, ..., K$. Because of the elaboration condition on goal networks with state-based transitions and because of the structure of goal networks, the invariants of the locations in a hybrid system with state-based transitions completely describe all transitions into and out of the locations. It is this property that is useful when trying to verify the hybrid system. Instead of finding all the transitions of the hybrid system, which can be prohibitive, the invariants could be used in the verification instead. This property also gives interesting results about the reachability of locations, which will be described in Section 4.4.

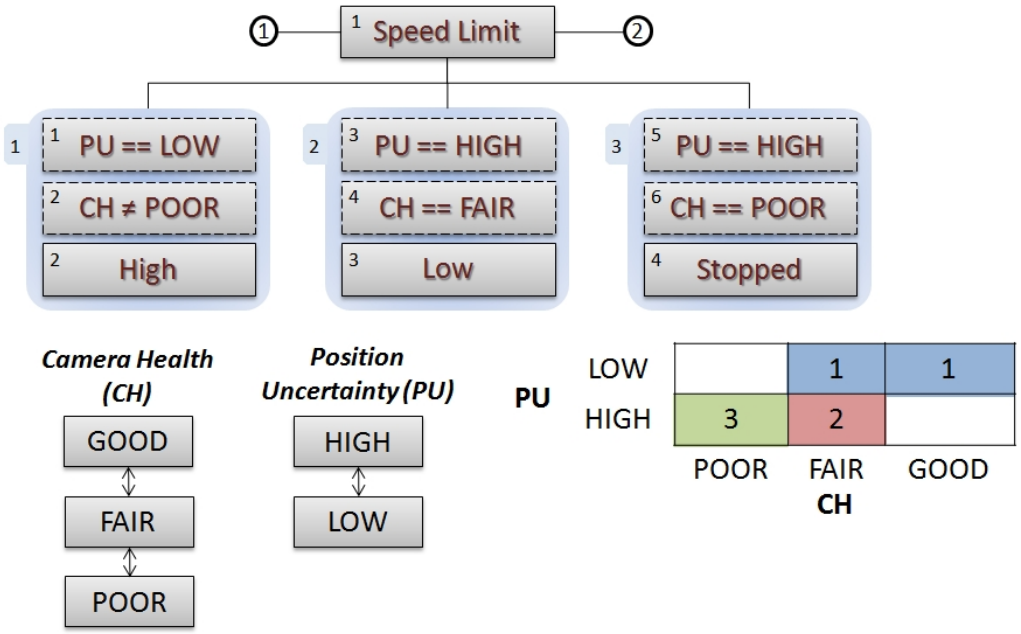


Figure 4.1: Goal tree that does not have state-based transitions with associated passive state models and passive state space

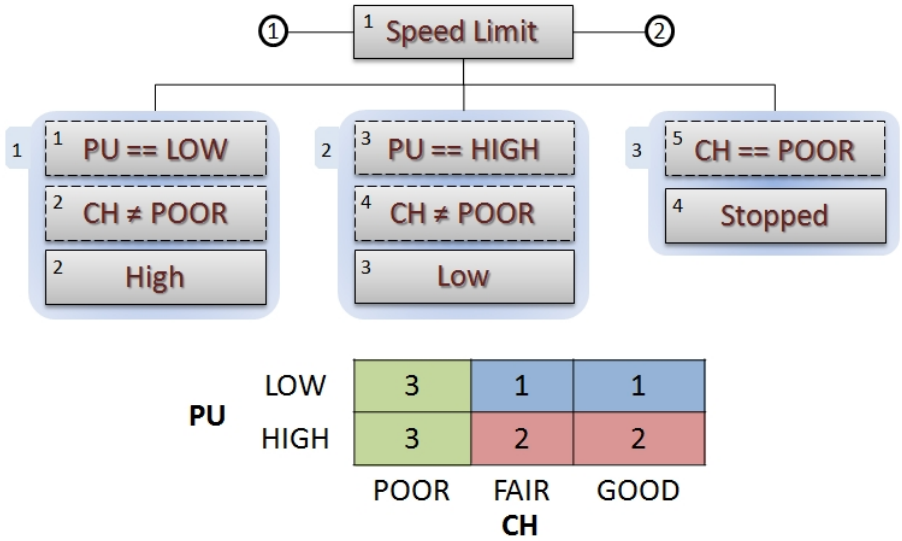


Figure 4.2: Goal tree with state-based transitions and associated passive state space

Therefore, goal networks that are designed to have state-based transitions can be verified using a very simple search algorithm that can handle complex systems.

While designing goal networks to have state-based transitions does impose some structure on the goal network, one could argue that the requirement is a good design practice. In the case where one or more passive states are not associated with an executable set of goals, it is unclear what would happen with the execution in that state. When state-based transitions are present, the goal network is maximally fault tolerant given the state model since every possible passive state is accounted for. In the case that there are passive states that satisfy the passive constraint of more than one executable set (or the invariant of more than one location), a non-deterministic execution scheme with weak fairness may be used and verified using the same method presented here.

4.3 SBT Checker

The conversion to hybrid systems and verification using symbolic model checkers presented in Chapter 3 is one approach to the verification of goal networks. However, the conversion procedure creates an automaton that captures all possible executions of the goal network; this can cause an explosion in the number of discrete modes (locations) relative to the numbers of goals and tactics present in the goal networks. While symbolic model checkers such as PHAVer handle large numbers of locations more easily than large numbers of state variables, there is still a limit. Abstractions and simplifications of the hybrid system can be used to aid in its verification, but another approach is to design the system for verification initially. To ensure that a goal network can be verified by the procedure described in this chapter, the goal network must have state-based transitions. For complex goal networks, this is not always easy to do by hand; therefore, the software program, SBT Checker, has been created to aid in the design process.

One way to verify that a goal network has state-based transitions is to convert it to a hybrid system and then compare each passive state to the locations' invariants in each group. However, for large systems with many locations and many states, this check can be time-consuming and ineffective for the iterative design process. Based on the following theorem, however, it is possible to check that the individual goal trees of each root goal in a goal network have state-based transitions, and that implies that the goal network has state-based transitions.

Let $\mathcal{D}_k$ be the set of state variables constrained by the passive goals in group U_k. Let $R_k = \{g_{r_1}^{0,0}, g_{r_2}^{0,0}, ..., g_{r_N}^{0,0}\}$ be the set of root goals that are in or have children in group $\mathcal{G}_k$. Let $\mathcal{S}_{r,k}$ be the

set of descendants of $g_r^{0,0}$, including passive goals and the root goal itself.

$$\mathcal{G}_k \subseteq \bigcup_{g_r^{0,0} \in R_k} \mathcal{S}_{r,k}, \tag{4.1}$$

because there may be extra root goals in $\mathcal{S}_{r,k}$. Let $\mathcal{D}_{r,k} = \{\text{svc}(u_m^{i_m,j_m}) | u_m^{i_m,j_m} \in \mathcal{S}_{r,k}\}$ be a set of all state variables constrained passively in $\mathcal{S}_{r,k}$. For some $\mathcal{D}_{r,k} = \{d_{n_1}, d_{n_2}, ..., d_{n_D}\}$, let $\Gamma_{r,k} = \Lambda_{n_1} \times \Lambda_{n_2} \times ... \times \Lambda_{n_D}$ be the passive state space of $\mathcal{S}_{r,k}$. Let $\gamma_i^r \in \Gamma_{r,k}$ be a passive state of the state variables in $\mathcal{D}_{r,k}$.

Let $\mathcal{L}_{r,k}$ be the set of executable branches of goals in $\mathcal{S}_{r,k}$. An executable branch of goals $L_j \in \mathcal{L}_{r,k}$ has the following properties:

1. All goals $g_n^{i_n,j_n} \in L_j$ are also in $\mathcal{S}_{r,k} \cap \mathcal{G}_k$.
2. If $g_n^{i_n,j_n} \in L_j$, its parent is also in L_j, $g_{i_n}^{i_{i_n},j_{i_n}} \in L_j$.
3. If $g_n^{i_n,j_n} \in L_j$, all its siblings are also in L_j.
4. If $g_n^{i_n,j_n} \in L_j$ and $g_n^{i_n,j_n}$ has at least one child goal in $\mathcal{S}_{r,k}$, then at least one child goal of $g_n^{i_n,j_n}$, $g_m^{i_m,j_m} \in \mathcal{S}_{r,k}$, $i_m = n$, is in L_j, $g_m^{i_m,j_m} \in L_j$.
5. All goals in L_j are compatible.
6. All goals in L_j are consistent.

Lemma 4.3.1. *For each pair of executable branches from different root goals, $L_i \in \mathcal{L}_{r_i,k}$, $L_j \in \mathcal{L}_{r_j,k}$, $r_i \neq r_j$, L_i is compatible with L_j.*

Proof. The general idea is that since the executable branches are drawn from different root goals, there are no shared parent goals across the executable branches. Assume that $L_i \in \mathcal{L}_{r_i,k}$ is incompatible with $L_j \in \mathcal{L}_{r_j,k}$, $r_i \neq r_j$. This means that there exists some $g_n^{i_n,j_n} \in L_i$ and $g_m^{i_m,j_m} \in L_j$ that have the same parent, $i_n = i_m$. Since all parent goals of each goal in an executable branch are also in that set by definition, this means that L_i and L_j have the same root goal, which negates the original assumption that $r_i \neq r_j$. □

Lemma 4.3.1 shows that executable branches from different root goals are compatible, and so they can be combined. The proposition introduced next says that if executable branch combinations are consistent, they are equivalent to executable sets of goals.

Proposition 4.3.2. *Let $\Upsilon_k = \mathcal{L}_{r_1,k} \times \mathcal{L}_{r_2,k} \times ... \times \mathcal{L}_{r_N,k}$ be the set of all combinations of executable branches from each root goal's set of goals. Let $\upsilon \in \Upsilon_k$; if all goals in υ are consistent, $\upsilon \equiv \theta_j$ for some $\theta_j \in \Theta_k$. Moreover, let $\Upsilon'_k = \{\upsilon | \upsilon$ is consistent$\}$. Then, $\Upsilon'_k \equiv \Theta_k$.*

Proof. Because Θ_k contains all possible executable sets in $\mathcal{G}_k$, if υ satisfies the definition of an executable set, there exists some $\theta_j \in \Theta_k$ such that $\upsilon \equiv \theta_j$. By the definition of the executable branches $L \in \mathcal{L}_{r,k}$, properties 1 and 3-5 of the executable set specification in Definition 3.4.3 are satisfied. All goals in each L are also in $\mathcal{G}_k$ by definition, so all the goals in the composition, $g_n^{i_n,j_n} \in \upsilon$ are also in $\mathcal{G}_k$ (property 1). Likewise, all parent goals, sibling goals, and at least one child goal are represented in each branch, so the composition of these branches originating from different root goals will have the same properties (properties 3–5). Property 6 of executable sets is satisfied by the composition of Υ_k; since all root goals with children in group $\mathcal{G}_k$ are represented by a set $\mathcal{L}_{r,k}$, at least one goal from each root goal with children in the group is represented in each υ. Property 2 is satisfied in a similar manner; if the root goal $g_r^{0,0} \in \mathcal{G}_k$, then for each $L \in \mathcal{L}_{r,k}$, $g_r^{0,0} \in L$ because of the parent goal requirement in the definition of L. Property 7 is satisfied by Lemma 4.3.1 and Property 8 is satisfied by the assumption above. Therefore, $\upsilon \equiv \theta_j$ for some $\theta_j \in \Theta_k$.

The above result can be applied to every $\upsilon \in \Upsilon'_k$ since each υ in that set is consistent. So, to prove that $\Upsilon'_k \equiv \Theta_k$, assume that there exists some $\theta_j \in \Theta_k$ such that there is not a corresponding $\upsilon \in \Upsilon'_k$. By definition, each goal in θ_j is either a root goal in $\mathcal{G}_k$ or descended from a root goal that has child goals in $\mathcal{G}_k$. Also by definition, each of those goals are present in a branch in the corresponding root goal's branch set, $\mathcal{L}_{r,k}$. Since there exists a set $\upsilon \in \Upsilon'_k$ that corresponds with every consistent combination of branches from each root goal $g_r^{0,0} \in R_k$, the executable set must either be missing a branch from at least one root goal's set or have more than one branch from at least one root goal's set. However, missing a branch from a root goal's set would violate either property 2 or 6 from Definition 3.4.3 of executable sets since either a root goal or descendants from a root goal would be missing from θ_j; so, θ_j must have two or more branches from at least one root goal's set. Let both $L_n \in \mathcal{L}_{r,k}$ and $L_m \in \mathcal{L}_{r,k}$ be subsets of θ_j; this implies that L_n and L_m are compatible. There must be some goals $g_n^{i_n,j_n} \in L_n$ and $g_m^{i_m,j_m} \in L_m$ such that $g_n^{i_n,j_n} \notin L_m$ and $g_m^{i_m,j_m} \notin L_n$ because one branch cannot be a subset of another by the definition of executable branches. The goals, $g_n^{i_n,j_n}$ and $g_m^{i_m,j_m}$, cannot be siblings and one cannot be the ancestor of the other by definition. However, they do have the same ancestor because they are in the same root

goal set. So, these goals must be incompatible by definition. Therefore, each executable set θ_j must include one and only one branch from each root goal and therefore there is a set $\upsilon \equiv \theta_j$ for each executable set $\theta_j \in \Theta_k$. Therefore, $\Theta_k \equiv \Upsilon'_k$. □

Theorem 4.3.3. *If for all controlled goals in branches from different root goals, $g_n^{i_n,j_n} \in \mathcal{L}_{r_n,k}$, $g_m^{i_m,j_m} \in \mathcal{L}_{r_m,k}$, $r_m \neq r_n$, the goals are consistent, $c(g_n^{i_n,j_n}, g_m^{i_m,j_m})$, and for all $g_r^{0,0} \in R_k$, the set of executable branches, $\mathcal{L}_{r,k}$, has state-based transitions over $\mathcal{D}_{r,k}$, then Θ_k has state-based transitions over $\mathcal{D}_k$.*

Proof. By definition, $\Upsilon_k = \mathcal{L}_{r_1,k} \times ... \times \mathcal{L}_{r_N,k}$ for all $g_{r_i}^{0,0} \in R_k$, $i = 1, ..., N$. By Proposition 4.3.2, the consistent subset, $\Upsilon'_k \subseteq \Upsilon_k$ is equivalent to Θ_k, so to prove this theorem, it is sufficient to show that Υ'_k has state-based transitions over $\mathcal{D}_k$. Let $I_k = \Upsilon_k \setminus \Upsilon'_k$ be the set of executable branch combinations with inconsistent goals. Since all active goals are consistent by assumption, each $\upsilon \in I_k$ has inconsistent passive goals only.

Let each $\mathcal{L}_{r,k}$ have state-based transitions over the corresponding passive state variable set $\mathcal{D}_{r,k}$, but assume that Υ'_k does not have state-based transitions over $\mathcal{D}_k$. That means that there exists some state $\gamma \in \Gamma_k$, where $\Gamma_k = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_D$, and D is the number of passive state variables in $\mathcal{D}_k$, such that there are no $\upsilon \in \Upsilon'_k$ such that $\gamma \models \text{pcons}(\upsilon)$. However, by the definition of state-based transitions, for each $\mathcal{L}_{r,k}$, there exists some $L_{r,\gamma} \in \mathcal{L}_{r,k}$ such that $\gamma \models \text{pcons}(L_{r,\gamma})$ because $\mathcal{L}_{r,k}$ has state-based transitions over $\mathcal{D}_{r,k} \subseteq \mathcal{D}_k$. By definition, there exists some $\upsilon \in \Upsilon_k$ such that

$$\upsilon = \bigcup_{g_r^{0,0} \in R_k} L_{r,\gamma}.$$

To satisfy the assumption that Υ'_k does not have state-based transitions, $\upsilon \in I_k$, which means that it has inconsistent passive goals. Without loss of generality, let the inconsistent goals be $u_n^{i_n,j_n}, u_m^{i_m,j_m} \in \upsilon$. By the definition of consistency, $\text{svc}(u_n^{i_n,j_n}) = \text{svc}(u_m^{i_m,j_m}) = d_i$ for some $d_i \in \mathcal{D}_k$ and for any $\lambda_j^i \in \Lambda_i$ such that $\lambda_j^i \models \text{cons}(u_n^{i_n,j_n})$, $\lambda_j^i \not\models \text{cons}(u_m^{i_m,j_m})$. Likewise, by the definition of γ, if any $\gamma \models \text{cons}(u_n^{i_n,j_n})$, then $\gamma \not\models \text{cons}(u_m^{i_m,j_m})$. Since $u_m^{i_m,j_m} \in \upsilon$, there must exist a $L_{r,\gamma} \subset \upsilon$ such that $u_m^{i_m,j_m} \in L_{r,\gamma}$. Then, $\gamma \not\models \text{pcons}(L_{r,\gamma})$, which means that $\mathcal{L}_{r,k}$ is not state-based and the original assumption is negated. □

The SBT Checker leverages this modularity of goal networks to check that each root goal's tactics have state-based transitions. The algorithm involves comparing the passive constraints in

Figure 4.3: Goal network for the state-based transitions verification example

Table 4.1: State Variable Data

State Variable	Abbreviation	Type
Position	X	Controlled
Camera Mode	CM	Controlled
Stabilizer Switch	SS	Controlled
Camera Health	CH	Passive
Position Uncertainty	PU	Passive
Vibrations	VB	Passive

each executable branch, $L_i \in \mathcal{L}_{r,k}$ for $g_r^{0,0} \in R_k$, to each passive state in the state space $\Gamma_{r,k}$ for the passive state variables in $\mathcal{D}_{r,k}$. Then, each passive state $\gamma \in \Gamma_{r,k}$ is checked against the passive constraints in each executable branch and if there exists some $\gamma \in \Gamma_{r,k}$ such that there is no $L_i \in \mathcal{L}_{r,k}$ where $\gamma \models \text{pcons}(L_i)$, those passive states are listed for the designer. The output of the SBT Checker software for the goal tree in Figure 4.1 would be $(\texttt{SH} == \texttt{GOOD} \wedge \texttt{PU} == \texttt{HIGH}) \vee (\texttt{SH} == \texttt{POOR} \wedge \texttt{PU} == \texttt{LOW})$. The output for the goal tree in Figure 4.2 would be `False`. The controlled goal consistency constraint is checked upon the goal network's conversion to a hybrid automaton in the verification software.

Because the number of executable sets, or locations, grows exponentially with the number of parent root goals, the modular approach saves computation time. It is also more conducive to an iterative and distributed design process since it gives nearly immediate feedback on the design of a root goal's goal tree.

A simple goal network example is shown in Figure 4.3. The robot's task is to drive to a point maintaining a safe velocity while taking pictures; Table 4.1 lists the state variables constrained in the goal network. The goal tree for the SpeedLimit goal is shown in Figure 4.2, and the goal tree for the TakePictures goal is shown in Figure 4.4. Both goal trees were verified to have state-based transitions by the SBT Checker which means that the entire goal network has state-based transitions. The goal network will be verified versus an unsafe set in the next section.

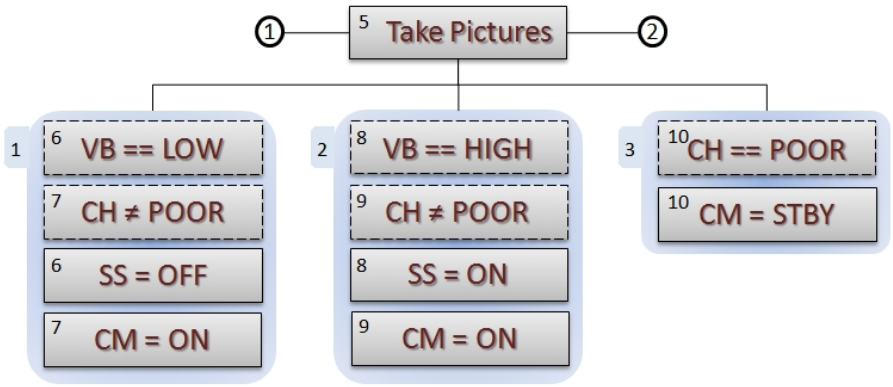


Figure 4.4: Goal tree of the TakePictures goal

4.4 InVeriant Verification Procedure

The idea behind the InVeriant software involves the special relationship between the invariant and the transition conditions for a hybrid system with state-based transitions. In a system with state-based transitions, if the state variables that are constrained in a group's locations have discrete states that are all reachable from each other, then each location in the group is reachable from any other location in the group. This is proved in Theorem 4.4.1 later. In this case, locations that satisfy the unsafe set are reachable if they exist. The InVeriant software creates the locations and invariants from the goal network and composes it with the unsafe set constraints to find unsafe locations.

While the assumption that the discrete states of the passively constrained state variables are reachable is usually a good one, there are times when it is not. In general, these passive state variables are health states or uncontrollable states of the environment that affect the way the system accomplishes a task. If there was a health or environment state value that was not reachable, it would not be modeled. However, continuous dependent state variables such as power or temperature may be constrained without knowing if a discrete set of states is reachable. These state variables, called continuous, rate-driven dependent state variables, have models whose discrete states do not match or correspond with the discrete sets of states that are passively constrained in the goal network. The discrete states in the model represent different rates of change of the state variable, which often depend on the controllable state variables, whereas the discrete sets of states constrained passively in the goal network depend on the continuous state space of the state variable. When unsafe locations constrain these state variables, their reachability must be confirmed by finding an appropriate path from the initial state to the unsafe state, a process that is aided by the state-based transition requirement.

The theorems that prove the methods used in the InVeriant software are presented next, followed by a formal treatment of the verification algorithm. The first theorem proves the reachability of locations introduced previously and the second theorem shows that within a set of locations that have the same discrete conditions on continuous, rate-driven dependent state variables, these locations are reachable.

Theorem 4.4.1. *Given a hybrid system with a set of locations $V_k = \{v_1, ..., v_n\}$ whose transitions are based on the discrete states of a set of passively-constrained state variables $\mathcal{D}_k = \{d_1, ..., d_m\}$, if all the discrete states associated with each passive state variable are reachable from each other, then all locations $v_l \in V_k$ are reachable from any other location, $v_j \in V_k$.*

Proof. Let V be a hybrid system with state-based transitions and let all discrete states $\Lambda_i = \{\lambda_1^i, ..., \lambda_{n_i}^i\}$ of each passive state variable $d_i \in \mathcal{D}$ be reachable from each other state, but assume location $v_l \in V_k$ is not reachable from $v_j \in V_k$. In order for v_l and v_j to be viable locations, they must have non-trivial invariants. That means that there must exist some $\gamma_j, \gamma_l \in \Gamma_k$ such that $\gamma_j \models \text{inv}(v_j)$ and $\gamma_l \models \text{inv}(v_l)$. Since the states of the passive state variables are all reachable, there is guaranteed to be a path from γ_j to γ_l. Let one possible path between the two system states be $p_{j,l} = \{\gamma_{i_1}, ..., \gamma_{i_n}\}$, where $p_{j,l}$ is the set of all intermediate states between γ_j and γ_l. Because V has state-based transitions, each $\gamma_i \in p_{j,l}$ has some $v_i \in V_k$ such that $\gamma_i \models \text{inv}(v_i)$. Since γ_j transitions to γ_{i_1} directly, there exists a transition condition $\tau_{j,i_1} = \text{inv}(v_{i_1})$. By induction through the path between states γ_j and γ_l, there is indeed a path between locations v_j and v_l, so the initial assumption is false. □

Theorem 4.4.2. *All locations $v_j \in V$ whose passive invariants have the same constraint, $\gamma^c \in \Gamma_k^c$, $\gamma^c \models inv(v_j)$, on the set of continuous dependent state variables, $\mathcal{D}_k^c$, are reachable from one another when all discrete passive state variables, $\mathcal{D}_k^d$ have reachable sets of states.*

Proof. Let $\mathcal{D}^c \subset \mathcal{D}$ be the set of all continuous, rate-driven dependent state variables and let

$$\Gamma^c = \prod_{d_i \in \mathcal{D}^c} \Lambda_i$$

be the state space of the passive constraints on those continuous, rate-driven dependent state variables. Likewise, let $\mathcal{D}^d \subset \mathcal{D}$ be the set of discrete passive state variables, $\mathcal{D}^d = \mathcal{D} \setminus \mathcal{D}^c$, and

let

$$\Gamma^d = \prod_{d_i \in \mathcal{D}^d} \Lambda_i$$

be the corresponding discrete passive state space. Let $V^{\gamma^c} \subset V$ be the set of locations satisfied by some state $\gamma^c \in \Gamma^c$, $V^{\gamma^c} = \{v_i | \gamma^c \models \text{inv}(v_i)\}$, where V has state-based transitions. Let $v_i, v_j \in V^{\gamma^c}$ and let v_j be unreachable from v_i. By the definition of state-based transitions, for all $\gamma \in \Gamma$ there exists some $v \in V$ such that $\gamma \models \text{inv}(v)$. It follows that for all $\gamma \in \Gamma^d_k \times \{\gamma^c\}$, there exists some $v \in V^{\gamma^c}$ such that $\gamma \models \text{inv}(v)$. Since all states $\gamma \in \Gamma^d \times \{\gamma^c\}$ are reachable from one another by design, it follows from Theorem 4.4.1 that $v_j \in V^{\gamma^c}$ is reachable from $v_i \in V^{\gamma^c}$, which contradicts the original assumption. □

The unsafe set is a collection of disjoint sets of constraints, $Z = \{\zeta_1, ..., \zeta_n\}$, where each disjoint set of constraints, $\zeta_i = \{z^i_1, ..., z^i_{n_i}\}$, has separate constraints on individual state variables, and each separate constraint $z \in (X^d \cup \dot{X}^c \cup \mathcal{D} \cup \dot{\mathcal{D}}^c) \times Q \times (\mathbb{R} \cup \Lambda)$ constrains a discrete controllable state variable (X^d), the rate of a continuous controllable state variable ($\dot{X}^c$), a passive state variable, or the rate of a continuous, rate-driven dependent state variable. The sets ζ of unsafe constraints are analogous to locations of the converted hybrid system, though the different sets in Z are not necessarily incompatible with one another; they are simply separate unsafe conditions against which the designer wishes to verify the system.

The verification algorithm goes through the following steps to verify a goal network versus the unsafe set, Z. A representation of this algorithm is shown in Figure 4.5.

1. Find all locations $v \in V$ from the goal network using the bisimulation procedure.

   (a) Find all potential executable sets of a goal network without checking for goal consistency. Find each location's passive invariant by listing all the passive constraints in that location; remove locations with inconsistent passive constraints.

   (b) Merge controlled constraints in each location and record any inconsistent controlled constraints. If there are any, stop and report which constraints are inconsistent. If not, continue.

2. For each $d_i \in \mathcal{D}$, the set of discrete states constrained passively in the goal network is Λ_i. Let $\mathcal{M}_i$ be the model of d_i, where $\mu^i_j \in \mathcal{M}_i$ is a discrete location in the model. For $d_i \in \mathcal{D}^d$, $\Lambda_i \equiv \mathcal{M}_i$. However, for $d_i \in \mathcal{D}^c$, the discrete sets are not always equivalent. Set $V' =$

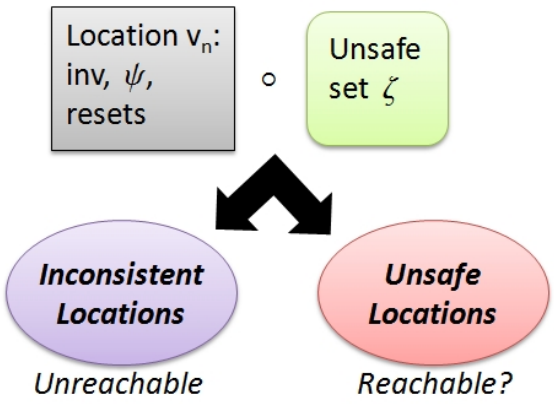


Figure 4.5: Representation of the InVeriant verification algorithm

V; for each $d_i \in \mathcal{D}^c$ such that $\Lambda_i \neq \mathcal{M}_i$, $V' = V' \circ \mathcal{M}_i = \{v_l \circ \mu_j | \forall v_l \in V, \forall \mu_j \in \mathcal{M}_i, v_l, \mu_j$ are consistent $\}$. A composed location $v' = v \circ \mu$ is defined as having a combined invariant, $\text{inv}(v') = \text{inv}(v) \wedge \text{inv}(\mu)$ and combined flow conditions, $\psi_{v'} = \psi_v \wedge \psi_\mu$.

3. For each $\zeta \in Z$:

   (a) Find the composition of the hybrid system and the unsafe set, $Y_\zeta = V' \circ \zeta = \{v_j \circ \zeta | \forall v_j \in V, v_j$ and ζ are consistent $\}$. Label all locations $y \in Y_\zeta$ as unsafe and output them.

   (b) Let the set of unsafe goals, $Y_{g,\zeta} = \{g_n^{i_n,j_n} \in y | \forall y \in Y_\zeta\}$, be the set of goals common to all unsafe locations. Output these goals.

   (c) Let the overloaded function cons() return the constrained value of a state variable when the function is given that state variable and a location (or set of constraints) as inputs. If there exists a $d_i \in \mathcal{D}^c$ such that $\text{cons}(d_i, y)$ exists, then a path must be found from $\text{init}(d_i)$ to $\text{cons}(d_i, y)$. There exists some $\lambda_j^i, \lambda_l^i \in \Lambda_i$ such that $\text{init}(d_i) \in \lambda_j^i$ and $\text{cons}(d_i, y) \cap \lambda_l^i \neq \emptyset$, where $\Lambda_i = \{\lambda_1^i, ..., \lambda_j^i, ..., \lambda_l^i, ..., \lambda_{n_i}^i\}$ is a forward or backward ordered set. Let $\Lambda_i^\zeta = \{\lambda_j^i, ..., \lambda_l^i\}$ be the set containing the two discrete sets of passively constrained values that satisfy the initial and unsafe constraints and all discrete sets of states in between. Then for each $\lambda_n^i \in \Lambda_i^\zeta$, a location $v \in V'$ must be found such that $(\lambda_n^i \models \text{inv}(v)) \wedge (\text{sign}(\text{cons}(\dot{d}_i, v)) = \text{sign}(\text{cons}(d_i, y) - \text{init}(d_i)))$ is true. If such a path can be found, the unsafe set is reachable.

This procedure is guaranteed by construction to find all locations in which unsafe conditions can occur. It can be applied to the example introduced in Section 4.3 as follows. Assume that the

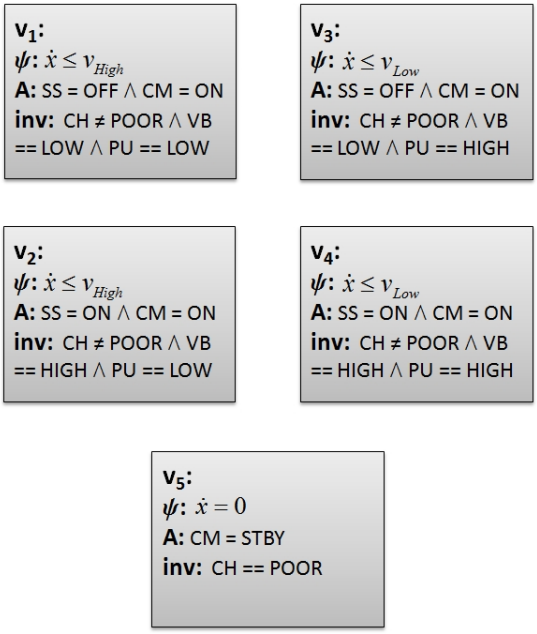


Figure 4.6: Converted locations, invariants, flow equations, and resets for the simple verification example

`StabilizerSwitch` state variable must always be in the `ON` position when the robot's velocity is high for safety reasons. Therefore, the unsafe set Z has one constraint, $\zeta = \{(\dot{\mathtt{X}}, >, v_{\text{low}}), (\mathtt{SS}, ==, \mathtt{ON})\}$. The first part of the InVeriant algorithm converted the goal network into a set of locations, invariants, flow equations, and resets, which are shown in Figure 4.6. Since none of the dependent state variables are continuous and rate-driven, the automaton did not need to be composed with any of the passive state model automata. So, the automaton in Figure 4.6 was composed with the unsafe condition ζ, and InVeriant found one location in which the unsafe set was reachable, v_1. The flow equations and reset equations of this location are consistent with the constraints in the unsafe condition. Since all of the passive states are reachable from each other passive state, this unsafe location is reachable. The combination of the first tactics in each goal tree are the culprits; these tactics must be redesigned so that they are inconsistent to be able to verify the goal network versus the given unsafe condition.

4.5 Verification of State and Completion-Based Linear Hybrid Systems

The software tools and verification method presented in the last two sections can also be applied to a broader class of linear hybrid automata. Essentially, the time point restrictions on the goal networks imposed for the conversion procedure induced a group structure on the linear hybrid automaton that resulted; this group structure, however, is not necessary for the verification method when starting from linear hybrid control systems. Lifting this requirement allows continuous controlled state variables to be reasoned about by InVeriant in the same way that continuous, rate-driven dependent state variables are. However, the state-based transition requirement must still hold for the overall (composed) hybrid system. Like in the case of the goal networks, it can be shown that the composition of hybrid automata that have state-based transitions also have state-based transitions if the dynamical flow constraints in the composed locations are consistent.

Definition 4.5.1. The dynamical flow constraints in locations of two hybrid automata are *consistent* if either they can be executed concurrently or there exists some rule that allows the constraints to be successfully merged into a composed dynamical flow constraint.

It is easy to see that this definition of consistency and the corresponding restriction are analogous to the goal network case of needing consistent controlled constraints. Often in the goal network case, the root goals would either be completion goals or would be macro goals combined with completion goals in the goal network. Since there is no good analog to the relationship that goal networks and goal trees have in the hybrid system case, a type of linear hybrid automaton called completion automata must be defined.

Definition 4.5.2. A *completion automaton* is a linear hybrid automaton in which the transition conditions and location invariants depend entirely on state variables that appear in the dynamical flow constraints or reset equations of the locations of that automaton.

Then, so-called state-based automata would have no transitions that depend on these types of state variables and instead would be automata that had state-based transitions as defined in the goal network case. Finally, the non-stochastic models of passive state variables would be called passive model automata. The composition of these types of automata is a hybrid system with certain properties as described by Theorem 4.5.3.

Theorem 4.5.3. *A hybrid automaton that is a composition of any number of completion and state-based automata has state-based transitions if all composed dynamical flow constraints are consistent.*

Proof. Let there be N state-based automata, H_n^p, where $n = 1, ..., N$. For each, the set of passive state variables constrained in the invariants of the locations is $\mathcal{D}_n \subseteq \mathcal{D}$. The composition of two automata $H' = H_i \circ H_j$ is defined as the joining of locations, $V' = V_i \circ V_j$, where each composed location $v' = v_i \circ v_j$, $v_i \in V_i$ and $v_j \in V_j$, has an invariant $\text{inv}(v') = \text{inv}(v_i) \wedge \text{inv}(v_j)$ and flow $\psi(v') = \psi(v_i) \wedge \psi(v_j)$. Assume that $H' = H_1^p \circ ... \circ H_N^p$, where H_n^p, $n = 1, ..., N$, have state-based transitions over $\mathcal{D}_n$, however, assume that H' does not have state-based transitions over

$$\mathcal{D} = \bigcup_{n=1}^{N} \mathcal{D}_n.$$

That means that there exists some passive state $\gamma \in \Gamma$, where Γ is the passive state space, such that for all $v \in V'$, $\gamma \not\models \text{inv}(v)$. However, since all H_n^p have state-based transitions over $\mathcal{D}_n \subseteq \mathcal{D}$, there exists some $v_n^\gamma \in V_n$ for all $n = 1, ..., N$ such that $\gamma \models \text{inv}(v_n^\gamma)$ by the definition of state-based transitions. The composition of all these locations, $v^\gamma = v_1^\gamma \circ ... \circ v_N^\gamma$, has an invariant

$$\text{inv}(v^\gamma) = \bigwedge_{n=1}^{N} \text{inv}(v_n^\gamma).$$

Since $\gamma \models \text{inv}(v_n^\gamma)$ for all $n = 1, ..., N$, by the linearity of the operator, $\gamma \models \text{inv}(v^\gamma)$. By the assumption, $\gamma \not\models \text{inv}(v)$ for all $v \in V'$, so v^γ must have an inconsistent invariant and so not be a location. However, if $\text{inv}(v^\gamma) = \text{False}$ and $\gamma \models \text{inv}(v^\gamma)$, then $\gamma = \text{False}$ and $\gamma \notin \Gamma$, which contradicts the original assumption. So, any number of state-based automata can be composed into an automaton that has state-based transitions.

By definition, completion automata, H_m^c, $m = 1, ..., M$, have invariants that depend only on controlled state variables, X. Since $X \cap \mathcal{D} = \emptyset$ and the transitions in the completion automata are also based on the locations' invariants, the invariants and transitions of these automata do not affect the passive state space. Let $H^c = H_1^c \circ ... \circ H_M^c$ be the composition of all completion automata. The composition, $v^\star = v^c \circ v'$, of any completion location $v^c \in V^c$ with any state-based location, $v' \in V'$, would have an invariant, $\text{inv}(v^\star) = \text{inv}(v^c) \wedge \text{inv}(v')$ that will never be inconsistent by definition since the invariants constrain different sets of state variables. Likewise, the dynamical flow constraints, $\psi(v^\star) = \psi(v^c) \wedge \psi(v')$, are consistent due to the assumption in

the theorem statement. So, no composed locations are inconsistent which means that since for all $\gamma \in \Gamma$, there exists some $v' \in V'$ such that $\gamma \models \text{inv}(v')$, the same will be true for all $v^\star \in V^\star$. Therefore, the composition of any number of completion and state-based automata will have state-based transitions, which proves the theorem. □

Because of this modularity, the SBT Checker can be used to check the state-based transitions of state-based automata. Likewise, the InVeriant verification software will check the consistent dynamical flow constraints upon composing the automata in preparation for model checking. Because the real requirement for InVeriant is a hybrid system whose invariant contains all the necessary information for the transitions of the system, if the composed completion automata have locations with unique invariants, this restriction is satisfied by that and the state-based transitions requirement. Continuous controlled state variables can then treated just like continuous, rate-driven dependent state variables (except no state variable model would need to be composed with the hybrid control system) in that a path must be found from the initial condition to the unsafe condition of those state variables to prove their reachability. Note that Theorems 4.4.1 and 4.4.2 still hold for the hybrid systems generated this way.

This result is important because it broadens the class of systems that can be verified using this method and it grants extra capabilities to the verification method by the addition of reasoning about continuous controllable state variables. This method is clearly more efficient than other symbolic model checkers such as PHAVer and HyTech that also deal with continuous states for systems that are designed with this additional structure. As discussed in the next section, at least some of the additional structure imposed is good design practice, so the SBT Checker and InVeriant verification toolbox may have many practical uses for the design and verification of real hybrid control systems.

4.6 Discussion

The verification procedure for systems that have state-based transitions has many positive attributes. First, the state-based transitions requirement can be checked modularly using a software tool, which allows for iterative and distributed design of control systems. The SBT Checker is a simple and intuitive tool that helps designers to design goal networks and hybrid automata that satisfy the state-based transitions requirement. This requirement forces certain structure in the goal network or hybrid automaton that aids in its verification; however, having state-based transitions is also a good design practice. When a system has state-based transitions, this means that every possible measured

passive state is accounted for and that there is a mode or tactic that can accommodate that state.

The InVeriant software is able to verify hybrid systems with state-based transitions quickly and efficiently. The structure imposed by the state-based transitions requirement allows the InVeriant software to ignore the individual transitions between hybrid system locations since all the necessary information about the transitions is contained in each location's invariant. The number of transitions between locations grows as the number of locations grow and as the number of passive state variables grow. Because the transitions are ignored, the complexity of the verification problem no longer depends on the number of state variables. In the verification method that involves the conversion from goal network to hybrid system followed by the use of a symbolic model checker, the number of passive state variables and particularly, the number of discrete states of each of those state variables contributed a large amount to the state space explosion of the system. Decoupling the passive state variables from the verification problem complexity will allow much larger systems to be verified using the InVeriant software.

The special structure of the systems that have state-based transitions allow for the reachability of the system to depend only on the states that the transitions constrain. Therefore, in many cases, a path to the unsafe locations does not need to be explicitly found. In the cases where a path is necessary, the structure of the system simplifies the process of finding a path since it can be shown that groups of locations are reachable from each other. Then, the path only needs to connect those groups. Rate constraints for continuous state variable can also be included in the unsafe set in InVeriant, which is not possible in many symbolic model checkers.

Because the InVeriant software was built to verify goal networks as well as hybrid systems, the information that it outputs is more conducive to the redesign of those goal networks. InVeriant will output not only the set of unsafe locations but also the goals and tactics that are common to all. PHAVer and HyTech output only the states of the state variables that allow the system to reach the unsafe set; even the unsafe locations are excluded from their output. Another benefit is not needing to translate the hybrid system into new notation or another language. Because the verification algorithm is so simple, there are only a few ways to reduce the complexity of the verification problem; either the number of locations in the hybrid system or the size of the unsafe set could be reduced. However, the modularity of the SBT Checker and the simplicity of the InVeriant verification algorithm suggest that this verification method could handle decently large goal network control programs or hybrid systems.

4.7 Conclusion

This verification method is designed for use with goal network control systems or hybrid systems that have state-based transitions. The state-based transitions requirement is a common sense restriction that leads to some very nice properties in the goal network verification. The modularity of the state-based transitions requirement makes the novel SBT Checker a useful design tool even for goal networks or hybrid systems that will not be verified. The InVeriant software leverages the reachability properties of the automaton imposed by the state-based transitions restriction and the properties of the state variables constrained to find the reachable unsafe set of the system quickly and efficiently. A significant example in Chapter 6 shows the speed of this verification method in relation to the conversion and PHAVer method described in Chapter 3. The simplicity and efficiency of this verification method suggest that it could be applicable to very large and complex systems, which is an important development in the use of reconfigurable goal-based control programs in autonomous robotic systems.

Chapter 5

Probabilistic Safety Analysis of Sensor-Driven Hybrid Automata

5.1 Introduction

The verification of control programs for autonomous robotic systems would be incomplete if the effect of state estimation uncertainty was not discussed. Autonomous systems see themselves and their environment through their sensors and estimators; all the state variables that drive the branching of the control program are, in fact, estimations of the true states. Because the control system is designed and verified for the actual states, an analysis of what may happen when the estimators are wrong is an important verification tool.

This chapter deals with the analysis of linear hybrid systems that have invariants and transition conditions that depend on the states of several uncontrollable and passively constrained state variables. Each uncontrollable state variable, which describes the environment or the health of the system (or anything that the system does not actively control), can reach some countable number of discrete states (or discrete sets of states) and the transitions between these states can be modeled as a stationary Markov process. These state variables' states are estimated by the system and these estimates of state are used by the linear hybrid automaton (LHA) to drive the discrete switching between locations. For a given LHA, there may be some combination of states and locations that are unsafe. In the perfect knowledge case (the estimator is always correct) when the LHA has been verified by a model checker, these unsafe conditions will never be met. However, when the accuracy of the estimator is not perfect, there exists some probability of reaching the unsafe condition. This chapter describes a method to calculate this failure probability for different types of LHA control systems.

It must be stated that the addition of uncertainty due to the estimation of the state variables driving the discrete transitions in the hybrid system adds some stochasticity to the problem. However, the problem is not effectively described as a stochastic hybrid system; as described later, it is better to think of the problem as an LHA with deterministic transitions and uncertainty. Stochastic hybrid systems include uncertainty in the transitions of the hybrid automaton as probabilistic transition conditions. Many methods to verify stochastic hybrid systems exist, some of which are referenced in Chapter 1. The method described here somewhat abstracts the transitions of the LHA, reducing the complexity of the problem dramatically.

The methods described in this chapter allow the analysis of the safety of a control program against the given unsafe set when the estimation of important state variables is not perfect. If these states were known exactly, a traditional hybrid system verification would be a sufficient test of the safety of this system. However, a full analysis of this system must include the calculation of the failure probability due to the estimation uncertainty. Section 5.2 sets up the failure probability calculation problem while Section 5.3 describes the steps of the calculation. Variations on this calculation procedure are discussed in Section 5.4. Because this method is very sensitive to problem complexity, reduction techniques are described in Section 5.5, followed by a discussion on methods to find an approximation of the failure probability in Section 5.6. Section 5.7 summarizes the contributions of this chapter.

5.2 Problem Definition

5.2.1 Automata Specification and Models

The architecture of the LHA control system involves a string of high-level completion tasks, such as driving a robot to a series of waypoints, that are executed in parallel with several minor tasks, such as maintaining the temperature of an instrument, limiting the robot's overall speed, or monitoring a sensor's health value. Thus, the locations in the hybrid automaton, v_i, can be sorted into disjoint groups, $V_1, V_2, ..., V_K$, based on which of the K high-level completion tasks that the location is trying to achieve. Then, each of the locations in a group V_k describes one method or tactic to complete the kth high-level task and all subtasks, and these tactics are chosen based on the states of the environment or uncontrollable states of the autonomous system that may affect the completion of the task. For example, the high-level task may be to drive the robot to a waypoint. One of the concurrent low-level tasks may be to maintain the temperature of the wheel motors above a

certain level through the use of two redundant strings of heaters. So, there are at least two tactics to accomplish the set of tasks; the first uses the primary set of heaters and the second uses the back-up heaters. The transition between these two locations is driven by the health of the primary string of heaters.

The flow of the linear hybrid automaton is

$$\psi_{i_f}(t_f)\tau_{i_f i_{f-1}}...\psi_{i_2}(t_2)\tau_{i_2 i_1}\psi_{i_1}(t_1)X_0 \tag{5.1}$$

where X_0 is the set of initial conditions on the controlled state variables, $\psi_{i_n}(t_n)$ is the flow associated with location v_{i_n} for t_n time steps, and $\tau_{i_n i_{n-1}}$ is the transition from location v_{i_n} to $v_{i_{n-1}}$. The flow of a location is based on the high-level and minor tasks that the location is trying to achieve. These dynamical equations are the continuous control actions that a particular tactic use in the completion of the appropriate task. Discrete control actions in the form of resets are grouped with the entry transition for a given location.

There are two types of transitions between the locations of an LHA with this group structure. First, transitions from a location, $v_i \in V_k$, to a location in the following group, $v_j \in V_{k+1}$, are called completion transitions, $\tau^c_{ji,k}$. The transition conditions in this case are based both on the state of the uncontrollable state variables of the system and on the completion of the high-level task that defines the group, V_k. The second type of transitions are transitions between locations in the same group, $v_i, v_j \in V_k$, and these are called failure transitions, $\tau^f_{ji,k}$. These transition conditions are between different tactics achieving the same high-level task and are based solely on the states of the uncontrollable state variables. Sometimes, the state of the system becomes such that there is no way to safely continue achieving a task; in that case, the automaton can transition from a location v_i to a special location called Safing and these failure transitions are labeled $\tau^f_{Si,k}$. All failure transition conditions must be entirely state-based; they cannot be based on the order of tactics attempted except in special circumstances described later. This restriction is not a serious one; in general, completely deterministic state-based transitions are a characteristic of more robust control programs. First, the definition of the complete system state of the uncertain state variables is given. Each of the uncontrollable state variables that is involved in failure transitions in the kth group of locations is called an uncertain state variable, $\chi \in \mathcal{U}_k$, where $\mathcal{U}_k$ is the set of all uncertain state variables in V_k.

Definition 5.2.1. A *complete system state*, s_j, is both the estimated and actual state values of each

uncertain state variable $\chi_i \in \mathcal{U}_k$ at a point in time in a possible execution of the LHA control system. The set of all possible complete system states is S. Each complete system state has two functions associated with it.

1. est$(s_j, \chi_i) \in \Lambda_i$ returns the estimated value of χ_i in s_j.

2. act$(s_j, \chi_i) \in \Lambda_i$ returns the actual value of χ_i in s_j.

Definition 5.2.2. For a hybrid automaton with *completely deterministic state-based transitions*, all states in a location's initial set, $s_i^\xi \in$ init(v_j), must satisfy the location's invariant, est$(s_i^\xi) \models$ inv(v_j). No transition $\tau_{jl,k}$ originating from the location can be satisfied by any states satisfying the location's invariant,

$$\text{est}(s_i^\xi) \models \text{inv}(v_j) \Rightarrow \text{est}(s_i^\xi) \not\models \tau_{jl,k},$$

for all $v_l \in V$, $v_l \neq v_j$. Also, for any location $v_j \in V_k$, there is only one transition $\tau_{jl,k}$ such that est$(s_i^\xi) \models \tau_{jl,k}$ for all $s_i^\xi \in \Xi_k$.

It is assumed that certain statistical information is known about the system. For each of the uncertain state variables, there must be a way to model the propagation of the state variable as a stationary Markov process; since these state variables are not controlled, oftentimes this approximation is a good one. Each state variable $\chi_i \in \mathcal{U}_k$ has a set $\Lambda_i = \{\lambda_1^i, \lambda_2^i, ..., \lambda_{n_i}^i\}$ of discrete state values (or discrete sets of state values) that can achieved by χ_i. The actual value of the state variable χ_i can be accessed by using its associated val$(\chi_i) \in \Lambda_i$ function. The probability of these state transitions, ρ_i are given by the following equation:

$$\rho_i(l, j) = P(\text{val}(\chi_i)[\kappa] = \lambda_j^i | \text{val}(\chi_i)[\kappa - 1] = \lambda_l^i), \tag{5.2}$$

for all $\chi_i \in \mathcal{U}_k$ and for all $\lambda_j^i, \lambda_l^i \in \Lambda_i$. The stationary probabilities give the probability that the state variable has a certain value in the steady state model, and these probabilities, α_i, are

$$\alpha_i(j) = P(\text{val}(\chi_i) = \lambda_j^i), \tag{5.3}$$

for all $\chi_i \in \mathcal{U}_k$ and for all $\lambda_j^i \in \Lambda_i$.

For each uncertain state variable, $\chi_i \in \mathcal{U}_k$, there exists an estimator for that state variable that has some non-zero amount of uncertainty. This uncertainty can be stated as a probability of

correctness of the estimated value. Furthermore, this probability can be broken up for each state value $\lambda_j^i \in \Lambda_i$ into the probability that if the value of the actual state, val(χ_i) is λ_l^i, then the value of the estimated state, val($\hat{\chi}_i$) is λ_j^i, for all $\lambda_j^i, \lambda_l^i \in \Lambda_i$. This probability, ρ_i^e, is

$$\rho_i^e(l, j) = P(\text{val}(\hat{\chi}_i)[\kappa] = \lambda_j^i | \text{val}(\chi_i)[\kappa] = \lambda_l^i). \tag{5.4}$$

The problems of measuring the uncertainty of an estimator and finding the probability of which state is estimated given an actual state are very real in practice. Methods such as simulation and testing can estimate these values; since the final failure probability should be used as a relative value, these probability values do not have to be exact. Much can be learned about the system by using these values as parameters that can be varied in the failure probability calculation.

5.2.2 Unsafe System States

When a LHA executes, that execution follows some path through the different locations based on, in this case, only the states of the system and time. For the automata described above, within a group the location that is executing is chosen based on the estimated system state alone; time only affects transitions out of groups. The execution path within a group, then, consists of a list of the estimated system state values and their associated locations at each time point; the length of the path is related to the completion time of the group, which will be defined in Section 5.2.3.

The conditions (states and locations) that should never occur in conjunction are called the unsafe set.

Definition 5.2.3. The *unsafe set* is a set of conditions $Z = \{\zeta_1, \zeta_2, ..., \zeta_n\}$ that should never be reached in an execution of the LHA. Each condition ζ_j contains a set of constraints on the uncontrollable state variables' values and a set of locations loc(ζ_j) $\subset V$ in which the set of constraints should not hold. This information can be accessed using the following functions:

1. plc(ζ_j, χ_i) $\subseteq \Lambda_i$ returns a set of discrete values that the state variable, χ_i, is constrained to be, val(χ_i) $\in$ plc(ζ_j, χ_i), to satisfy the unsafe condition. If χ_i does not affect this unsafe condition, plc(ζ_j, χ_i) $= \Lambda_i$.

2. loc(ζ_j, v_i) returns true if $v_i \in$ loc(ζ_j) and false otherwise.

In a verified system, the unsafe set is unreachable when the estimated system state is accurate; failure can occur, however, when state estimation uncertainty is added. While the execution path for

a group of locations depends only on the estimated system state at each time point, determining if an execution path reaches the unsafe set requires both the estimated and actual system states. The estimated system state still determines which location will be executing at a given time, and the combination of the executing location and the actual system state dictate entrance into the unsafe set. The lemma that follows states exactly this.

Definition 5.2.4. Each location $v_n \in V_k$ has a function associated with it that returns the state values that each uncertain state variable, $\chi_i \in \mathcal{U}_k$, must take in order for that location to be executed, $ucons(v_n, \chi_i) \subseteq \Lambda_i$. This is the location's passive invariant in the state-based transitions case. If a state variable χ_i is unconstrained in a location v_n, $\text{ucons}(v_n, \chi_i) = \Lambda_i$.

Lemma 5.2.5. *Let $\Omega_k \subset S$ be the set of complete system states that drive the automaton execution from group V_k into the unsafe set Z. For a complete system state $s^\omega \in \mathcal{S}$, $s^\omega \in \Omega_k$ if and only if there exists $\zeta_j \in Z$ and $v_n \in V_k$ such that*

$$\left(\bigwedge_{\chi_i \in \mathcal{U}_k} act(s^\omega, \chi_i) \in plc(\zeta_j, \chi_i) \right) \wedge \left(\bigwedge_{\chi_i \in \mathcal{U}_k} est(s^\omega, \chi_i) \in ucons(v_n, \chi_i) \right) \wedge loc(\zeta_j, v_n) \quad (5.5)$$

is true.

Proof. Let $s^\omega \in \Omega_k$ but assume there is no v_n that satisfies (5.5). By the definition of the unsafe set, there must exist some $\zeta_j \in Z$ such that

$$\bigwedge_{\chi_i \in \mathcal{U}_k} \text{act}(s^\omega, \chi_i) \in \text{plc}(\zeta_j, \chi_i)$$

is true, since entrance into the unsafe set is always driven by the actual system state. Since the unsafe set specifies the total system state, including the location in the automaton, there must exist some v_n such that $\text{loc}(\zeta_j, v_n)$ is true. To enter location v_n and thus the unsafe set from complete system state s^ω,

$$\bigwedge \text{est}(s^\omega, \chi_i) \in \text{ucons}(v_n, \chi_i)$$

must be true since the transitions in the automaton are state driven by definition and because the estimated state drives the transitions in the hybrid automaton execution. Therefore, v_n does satisfy (5.5). The other direction of the proof is obvious by the definition of the unsafe set. □

Complete system states that cause the automaton to transition from a location $v_i \in V_k$ to Safing

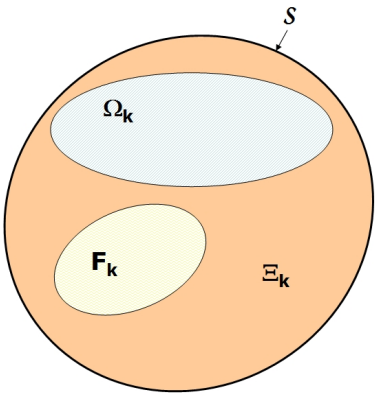


Figure 5.1: A representation of the classifications of complete system states. The nominal set of states is Ξ_k, which is $S_k \setminus (\Omega_k \cup F_k)$, where F_k is the set of Safing states and Ω_k is the set of unsafe states.

are elements of the safing set for group V_k, $s^f \in F_k$. Complete system states that allow the execution of the group to occur normally are elements of the nominal set, $s^\xi \in \Xi_k$. For each group of locations V_k,

$$S = \Xi_k \cup F_k \cup \Omega_k, \tag{5.6}$$

where S is the set of all complete system states, and the sets Ξ_k, F_k, and Ω_k are disjoint. Figure 5.1 illustrates this.

5.2.3 Failure Path Specification

The completion time of a group V_k depends on the completion task that defines the group. The type of completion time (uniform or non-uniform) depends on the presence of rate limiting tasks that affect the completion task in the group.

Definition 5.2.6. A *nominal execution path* of a group V_k is a path $s_1^\xi s_2^\xi ... s_r^\xi \in N_k$ in which only nominal complete system states are visited before the group is exited and execution continues in group V_{k+1}. The set of all nominal execution paths in group V_k is N_k.

Definition 5.2.7. Given the set of nominal execution paths N_k for group V_k, the *completion time*, c_k, is the minimum length of nominal execution path,

$$c_k = \min_{\nu \in N_k} \text{length}(\nu). \tag{5.7}$$

The completion time of a group is the time it takes to achieve the group's completion task at the fastest constrained rate.

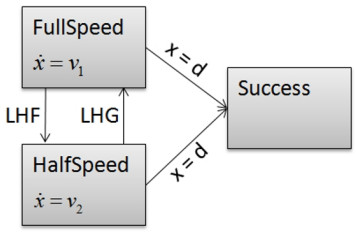


Figure 5.2: Hybrid control system for speed limit example

Definition 5.2.8. In a *uniform completion* group, V_k,

$$c_k = \min_{\nu \in N_k} \text{length}(\nu) = \max_{\nu \in N_k} \text{length}(\nu). \tag{5.8}$$

The uniform completion case holds in groups that have only one rate of completion of the task; in this case, all tactics contribute the same amount towards the completion of the task. Another way to define the uniform completion case is that the contribution values of each location in the group are the same.

Definition 5.2.9. The *contribution value* of a location $v_i \in V_k$, $\text{cval}(v_i) \in \mathbb{R}$, is the normalized contribution towards the achievement of the completion task in V_k that location v_i gives each time step. In uniform completion groups, for each $v_i \in V_k$, $\text{cval}(v_i) = 1$.

Definition 5.2.10. A *non-uniform completion* group is one in which

$$\min_{\nu \in N_k} \text{length}(\nu) \neq \max_{\nu \in N_k} \text{length}(\nu). \tag{5.9}$$

In the non-uniform completion, for each location $v_i \in V_k$, $\text{cval}(v_i) \leq 1$. In this case, the group would have more than one rate that constrains the achievement of the completion task. An example of non-uniform completion would be a rover that is assigned to reach a certain waypoint, but whose maximum velocity is related to the laser sensor's health value. One time step in a location would drive the rover a distance that is different than the distance achieved in a different location that constrains the rover to a different maximum velocity. Figure 5.2 shows the simple hybrid system for this example; assuming that $v_1 = 2v_2$, the contribution value of the first location would be 1 and the contribution of the second location would be $\frac{1}{2}$.

The definition of failure path is now given.

Definition 5.2.11. A *failure path* in group V_k is a sequence of nominal complete system states, s_i^{ξ},

$i = 0, ..., n$, followed by an unsafe system state, s^{ω}. The number of nominal complete system states is $n = 0, ..., r-1$, where $r = c_k$ in the uniform completion case and depends on the completion time and the contribution values of the locations visited along the path for the non-uniform completion case.

From Definition 5.2.2 of completely deterministic state-based transitions, it is clear that there can be a function cloc$(s^{\xi}, k) \in V_k$ that returns the location associated with the nominal complete system state s^{ξ} in group V_k.

Lemma 5.2.12. *For every failure path $\pi = s_1^{\xi} s_2^{\xi} ... s_{r-1}^{\xi} s^{\omega} \in \Pi_k$, where Π_k is the set of all failure paths in group V_k,*

$$\sum_{i=1}^{r-1} cval(cloc(s_i^{\xi}, k)) < c_k. \tag{5.10}$$

Proof. The proof of this lemma is simple; if the sum of the contribution values of the nominal states visited in a failure path equals or exceeds c_k, the execution continues into the next group by the definition of completion time. In order for the path to lead to the unsafe set within a group, the sum of the contribution values of the nominal states must ensure that the path is fully contained within the group; entrance into the unsafe set is always the last state transition in a failure path. □

Each failure path has some probability of occurring during an execution of that group of the hybrid automaton. This is called the failure path probability, and the sum of these over all possible failure paths is the failure probability of a group. Details of these calculations will be described in the next section.

5.3 Probability Calculations

The failure probability is calculated from the sum of all the failure path probabilities in a group; the paths in a group depend on the completion time, and the procedure for finding all the failure paths in a group depends on whether it is a uniform or non-uniform completion group. The failure paths for the uniform completion case are easy to find; the procedure for the non-uniform completion case is more involved. Both will be described in this section.

First, a combined Markov process-like transition probability matrix and a set of initial probabilities are calculated for each complete system state. The stationary probabilities of the individual Markov chains are used to calculate initial condition probabilities of each complete system state in

a group. By having this information, the separate group failure probabilities may be combined into a failure probability for the entire hybrid automaton, as will be shown later.

For a given complete system state, the initial probability, $P(s)$, is given by the following:

$$P(s) = \prod_{\chi_i \in \mathcal{U}_k} P(\text{val}(\chi_i) = \text{act}(s, \chi_i)) P(\text{val}(\hat{\chi}_i) = \text{est}(s, \chi_i) | \text{val}(\chi_i) = \text{act}(s, \chi_i)). \quad (5.11)$$

The transition probability of going from one complete system state to another, $P(s_l|s_j)$, is given by the following:

$$P(s_l|s_j) = \prod_{\chi_i \in \mathcal{U}_k} P(\text{val}(\chi_i)[\kappa] = \text{act}(s_l, \chi_i) | \text{val}(\chi_i)[\kappa - 1] = \text{act}(s_j, \chi_i)) \times$$
$$P(\text{val}(\hat{\chi}_i)[\kappa] = \text{est}(s_l, \chi_i) | \text{val}(\chi_i)[\kappa] = \text{act}(s_l, \chi_i)). \quad (5.12)$$

5.3.1 Uniform Completion Case

For the uniform completion case, collections of stationary and transition probabilities between nominal and unsafe system states can be created. First, the probability of executing a failure path of length one in group V_k (i.e., starting in the unsafe set) is

$$a_k = \sum_{s_j^\omega \in \Omega_k} P(s_j^\omega). \quad (5.13)$$

Likewise, let W_k be a vector of probabilities whose elements are the initial probabilities of each nominal system state, $s_j^\xi \in \Xi_k$. Thus, the jth element of vector W_k is

$$W_k(j) = P(s_j^\xi). \quad (5.14)$$

Next, transition probability constructs are defined. The matrix of transition probabilities between all nominal complete system states is Q_k, where the probability of a transition between s_i^ξ to s_j^ξ is

$$Q_k(i, j) = P(s_j^\xi | s_i^\xi). \quad (5.15)$$

Since all unsafe complete system states are accepting states, only the transitions into these states from the nominal complete system states are considered. The vector $W_{u,k}$ contains the probabilities

of transitions from each nominal complete system state to every unsafe complete system state,

$$W_{u,k}(j) = \sum_{s_i^\omega \in \Omega_k} P(s_i^\omega | s_j^\xi). \tag{5.16}$$

Proposition 5.3.1. *The failure probability for the uniform completion case in group V_k can be calculated using the following formula, for $c_k \in [2, \infty)$,*

$$W_s(k) = a_k + W_k \cdot (\sum_{i=0}^{c_k-2} Q_k^i) W_{u,k}. \tag{5.17}$$

When $c_k \to \infty$, the equation becomes

$$W_s(k) = a_k + W_k \cdot (I - Q_k)^{-1} W_{u,k}. \tag{5.18}$$

Proof. The failure probability is the sum of all the failure path probabilities; for the uniform completion case and the definitions of a_k, W_k, Q_k, and $W_{u,k}$ given above, Eq. (5.17) sums the path probabilities of all failure paths of length one to length c_k. If a path has length $c_k + 1$, c_k of the path elements must be nominal states; because for the uniform completion case, cval$(s^\xi) = 1$ for all $s^\xi \in \Xi_k$ and by Lemma 5.2.12, a path of length $c_k + 1$ is not possible in group V_k. Therefore, Eq. (5.17) is the sum of all possible failure paths in V_k. Using

$$\sum_{i=0}^{\infty} Q^i = (I - Q)^{-1}, \tag{5.19}$$

one can derive (5.18) from (5.17). □

In the infinite time case, the failure probability approaches $1 - W_f(k)$, where $W_f(k)$ is the probability of entering the Safing location from group V_k.

5.3.2 Non-Uniform Completion Case

In the uniform completion case, the number of failure paths considered was greatly reduced by the creation of the probabilistic transition matrix and vectors. Since all locations contributed the same amount to the completion of the group, the path length did not depend on which individual locations were visited. This is not the case in the non-uniform completion case, where the execution

path length does depend on which locations are executed and in what order. Like the uniform completion case, there is a way to reduce the number of failure paths that must be considered by grouping together locations by contribution values.

Let $B_k = \{b | b = \text{cval}(v_i), \forall v_i \in V_k\}$ be the set of contribution values in a group, where all $b \in B_k$ are unique (for all $b_i, b_j \in B_k, b_i \neq b_j$) and ordered ($B_k = \{b_1, b_2, ..., b_n\}$ such that $b_1 > b_2 > ... > b_n$). Since cval(v_i) is the rate of location $v_i \in V_k$ normalized by the maximum rate in group V_k, $b_1 = 1$. Now, let there be n sets β_i, where

$$\beta_i = \{s_j^\xi | \text{cloc}(s_j^\xi, k) = v_i \wedge \text{cval}(v_i) = b_i, \forall s_j^\xi \in \Xi_k\} \tag{5.20}$$

where each β_i is the set of locations that have a contribution value b_i. Then, failure paths can be created using β_i instead of using the individual nominal system states, s^ξ, where for failure path $\beta_{i_1}\beta_{i_2}...\beta_{i_{r-1}}s_r^\omega$,

$$\sum_{j=1}^{r-1} b_{i_j} < c_k. \tag{5.21}$$

All possible failure paths can be found using a simple algorithm that is based on a breadth-first search. A branch of the search tree is complete when adding any set β_i, $i = 1, ..., n$, to the path, π, would cause the sum of the contribution values of the path elements to equal or exceed the completion time,

$$\sum_{\beta_i \in \pi} b_i \geq c_k - b_n. \tag{5.22}$$

Also, the paths as they stand at each level of the search are added to the set Π of all potential failure paths. The algorithm is outlined below:

1. Initialize the search tree with the initial failure path placeholder, $\beta_\emptyset$. The initial failure path is the failure path with no nominal transitions. The placeholder, $\beta_\emptyset$, has no contribution value and is ignored in any path containing other β sets. Add this path to set Π_1. Set the level counter $l = 1$.

2. For each branch, π_i^l, on level l, compute the sum of the contribution values,

$$\text{cval}(\pi_i^l) = \sum_{\beta_j \in \pi_i^l} b_j. \tag{5.23}$$

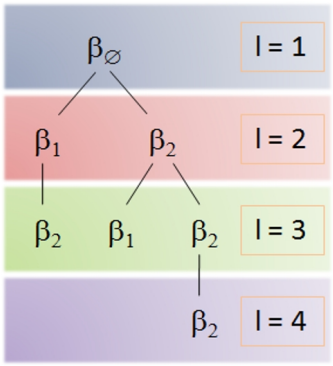


Figure 5.3: The search tree of potential failure paths for the speed limit example

(a) For each $b_j \in B_k$, if $\text{cval}(\pi_i^l) + b_j < c_k$, append β_j to the path, $\pi_i^l + \beta_j \in \Pi_{l+1}$.

(b) If $b_n = 0$, for each path π_i^l whose last set is β_n, β_n may not be added to that path.

3. Increment l.

4. Repeat steps 2 and 3 until $\Pi_l = \emptyset$.

5. Create the set of all potential failure paths,

$$\Pi = \bigcup_{i=1}^{l-1} \Pi_i. \tag{5.24}$$

This algorithm can be demonstrated with the simple speed limit example shown in Figure 5.2. Let $c_k = 2$ and $B = \{1, \frac{1}{2}\}$, where the location with the higher speed limit has a velocity constraint twice the other location's. The breadth-first search tree can be found in Figure 5.3 with the search levels denoted. The final set of potential failure paths is $\Pi = \{\beta_\emptyset, \beta_1, \beta_2, \beta_1\beta_2, \beta_2\beta_1, \beta_2\beta_2, \beta_2\beta_2\beta_2\}$. Each path in Π would need to be followed by a transition into the unsafe set for it to become a failure path; the element $\beta_\emptyset$ would simply be replaced by an initial condition in the unsafe set.

In the case that $b_n = 0$, the failure path algorithm has an exception. Because the execution can remain in the zero rate locations for an infinite number of time steps, once β_n is added to a path, the next set added to the path, β_j, must have a contribution value $b_j > 0$. This step avoids the path creation algorithm from becoming an infinite loop; the infinite number of failure paths is accounted for in the path probability calculation step.

The calculation of failure path probabilities is similar to the method described for the uniform completion case. The scalar value a_k is again the probability of starting in the unsafe set. Instead

of one vector of initial probabilities for the nominal states, W_k, there are n vectors, one for each set β_i, $i = 1, ..., n$. Each vector W_k^i has an initial probability for each nominal complete system state $s_j^\xi \in \beta_i$, calculated in (5.14). The Q_k transition matrix is broken up into several smaller matrices to account for all possible transitions between the β sets. For all β_i, β_j, $i, j = 1, ..., n$, in a group, there exists a matrix $Q_k^{i,j}$ that contains the transition probabilities from each nominal complete system state $s_l^\xi \in \beta_i$ to each $s_m^\xi \in \beta_j$. Finally, the vector of transition probabilities from nominal complete system states to the unsafe set, $W_{u,k}$ is also broken up into n vectors $W_{u,k}^i$, $i = 1, ..., n$, that are associated with the β sets.

The failure path probabilities are calculated using the initial and transition vectors and matrices described above and then all path probabilities are summed to find the group's failure probability. For the speed limit example, there are seven failure paths whose individual path probabilities must be calculated and summed to find the group's failure probability, as follows:

$$W_s(1) = a_1 + W_1^1 \cdot W_{u,1}^1 + W_1^2 \cdot W_{u,1}^2 + W_1^1 \cdot (Q_1^{1,2} W_{u,1}^2) + W_1^2 \cdot (Q_1^{2,1} W_{u,1}^1) + W_1^2 \cdot (Q_1^{2,2} W_{u,1}^2) + W_1^2 \cdot (Q_1^{2,2} Q_1^{2,2} W_{u,1}^2). \quad (5.25)$$

In the case that $b_n = 0$, each time a transition into β_n is encountered in a failure path, it is replaced by an infinite series of paths with increasing numbers of transitions into that set. Whereas a single transition into set β_j, $b_j > 0$ from set β_i is generally indicated in the failure path probability calculation by $Q_k^{i,j}$, a transition from β_i to β_n, $b_n = 0$ would be represented by

$$Q_k^{i,n} \left(\sum_{x=0}^{\infty} \left(Q_k^{n,n} \right)^x \right). \quad (5.26)$$

By (5.19), this becomes $Q_k^{i,n} \left(I - Q_k^{n,n} \right)^{-1}$. Likewise, if β_n is the first set of locations visited, its probability value is represented by $W_k^n \cdot \left(I - Q_k^{n,n} \right)^{-1}$.

5.3.3 System Failure Probability

The overall hybrid automata failure probability can be calculated by summing the probabilities of all the failure paths through the automata. To do this, the probability of each group reaching the Safing location, $W_f(k)$, must be calculated. The procedure for doing this is the same as for finding the failure probability, however, $s_i^f \in F_k$ are used as the accepting states instead of $s_i^\omega \in \Omega_k$. The

probability of traversing group V_k nominally is then

$$W_n(k) = 1 - W_s(k) - W_f(k). \tag{5.27}$$

Proposition 5.3.2. *The failure probability of the system of $K > 1$ groups is given by*

$$W_s = W_s(1) + \sum_{i=2}^{K}(\prod_{j=1}^{i-1} W_n(j))W_s(i). \tag{5.28}$$

Proof. The failure probability of the hybrid system is the sum of the failure path probabilities through the system. Since the failure paths can only consist of zero to $K-1$ nominal group transitions followed by a failure, Eq. (5.28) gives all failure paths through the hybrid system. Any entrance into Safing removes the execution from the hybrid system and precludes failure in the future, and so is excluded from the failure probability calculation. □

5.4 Variations on the Failure Probability Problem

5.4.1 Subgroups

Some groups may have two or more disjoint sets of locations; once execution enters one of the sets of locations in group V_k, the execution can only exit that set of locations to go to Safing or the next group, V_{k+1}. These disjoint sets of locations are called subgroups. Each subgroup, $V_{k,h} \subset V_k$, has a set $I_{k,h} \subset \Xi_k$ of complete system states that cause the execution of the automaton to enter the subgroup $V_{k,h}$ upon entering the group V_k. The initial transition into a subgroup allows only a subset of all nominal complete system states, however once in a subgroup, every complete system state continues the execution in that subgroup. Therefore, the W_k probability vector must be broken up into separate, disjoint $W_{k,h}$ vectors for each subgroup. Once the execution enters a subgroup, the unsafe set of complete system states may be different between the subgroups and from the initial set of unsafe complete system states; therefore, each subgroup has its own $F_{k,h}$, $\Xi_{k,h}$, and $\Omega_{k,h}$ for the non-initial transitions within the subgroup. This triggers separate transition probability matrices and nominal to unsafe transition vectors for each subgroup as well.

An example of this subgroup structure can be found in the following example. Suppose that there is a rover that must follow a path to a point, but the path branches and the choice of the final point is based on the rover's system health. Figure 5.4 shows the task and Figure 5.5 gives

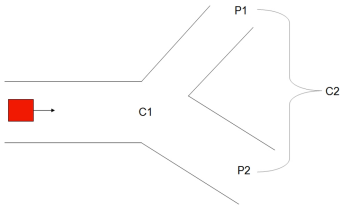


Figure 5.4: Depiction of the path for the simple rover task

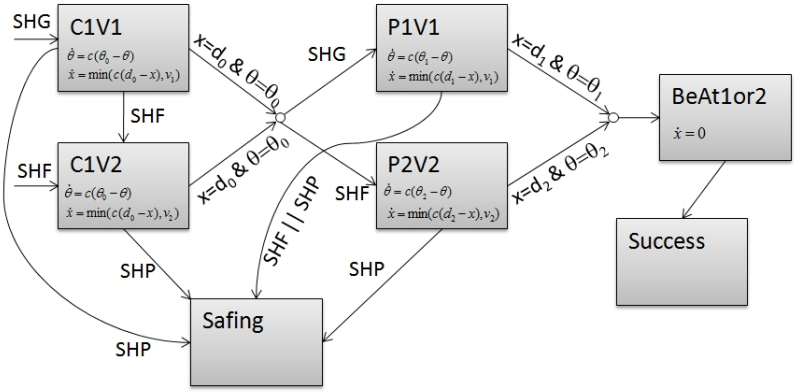


Figure 5.5: Hybrid control system for the simple rover task

the hybrid control system. Once the robot reaches point C1 and makes the choice to go to P1 or P2, this choice cannot be reversed. The set of all complete system states for this task is $S = \{\texttt{GG}, \texttt{GF}, \texttt{GP}, \texttt{FG}, \texttt{FF}, \texttt{FP}, \texttt{PG}, \texttt{PF}, \texttt{PP}\}$ (actual then estimated system health state), and the unsafe set is as follows:

1. $\dot{x} > 0$ and `SystemHealth` is `POOR`, and
2. $\dot{x} > v_2$ and `SystemHealth` is `FAIR`.

The initial unsafe set is $\Omega_2 = \{\texttt{FG}, \texttt{PG}, \texttt{PF}\}$. The initial set that allows transitions into the top location (going to P1) is $I_{2,1} = \{\texttt{GG}\}$; the initial set for the other location (going to P2) is $I_{2,2} = \{\texttt{GF}, \texttt{FF}\}$. The remaining complete system states are initially Safing, $F_2 = \{\texttt{GP}, \texttt{FP}, \texttt{PP}\}$.

Once the execution transitions into the first subgroup, all complete system states can be reached, and the unsafe, safing, and nominal sets for this subgroup are as follows: $\Omega_{2,1} = \{\texttt{FG}, \texttt{PG}\}$, $F_{2,1} = \{\texttt{GF}, \texttt{GP}, \texttt{FF}, \texttt{FP}, \texttt{PF}, \texttt{PP}\}$, and $\Xi_{2,1} = \{\texttt{GG}\}$. For the second subgroup, these sets are $\Omega_{2,2} = \{\texttt{PG}, \texttt{PF}\}$, $F_{2,2} = \{\texttt{GP}, \texttt{FP}, \texttt{PP}\}$, and $\Xi_{2,2} = \{\texttt{GG}, \texttt{GF}, \texttt{FG}, \texttt{FF}\}$.

Though a group may be have non-uniform completion, each subgroup within that group may be uniform or non-uniform. In the example above, group V_2 as a whole is a non-uniform completion group, but each subgroup, $V_{2,1}$ and $V_{2,2}$, is a uniform completion group. Once the initial transition

into a subgroup is taken, the subgroup can be treated like any other group when calculating the failure probability within the subgroup. These subgroup failure probabilities, $W_s(k,h)$, are calculated in the same way as the group failure probability for a connected group except that there is no initial failure probability (a_k) in the failure path probability sum. Then, the overall group failure probability for a group that has H subgroups is the sum of the initial failure probability and all the subgroup failure probabilities,

$$W_s(k) = a_k + \sum_{h=1}^{H} W_s(k,h). \tag{5.29}$$

The safing probability, $W_f(k)$, for a group V_k with subgroups is calculated in the same way; the nominal probability is $W_n(k) = 1 - (W_s(k) + W_f(k))$. These group probabilities can then be used to calculate the system failure probability as described in Eq. (5.28).

5.4.2 Completion Time Uncertainty

It may not be possible to exactly know the completion time for a group. If there is a probability distribution over a finite number of possible completion times, the failure probability for that group can be calculated by finding the failure probability for each possible completion time. The total failure probability for the group is the sum of the failure probabilities for each possible completion time multiplied by the completion time probability. This procedure works for both the uniform and non-uniform completion cases.

Let H be a hybrid automaton with stochastic differential equations,

$$dx_c = l(v, x_c)dt + \sigma(v, x_c)dw \tag{5.30}$$

that dictate the controllable state evolution for the completion task in each location. The drift, $l : V \times X_c \rightarrow \mathbb{R}$ is the linear flow condition for the continuous completion state variable and $\sigma : V \times X_c \rightarrow \mathbb{R}^{1\times p}$ is a dispersion vector for the $\mathbb{R}^p$-valued Wiener process $w(t)$. It is assumed that for any $v_i, v_j \in V_k$, $\sigma(v_i, x_c) = \sigma(v_j, x_c)$. For each group, V_k, let $L_k = \{l(v, x_c)|\forall v \in V_k\}$, the set of all completion rates in a group. Let $l_k = \min_{l_i \in L_k} l_i$; the completion time,

$$c_k = \frac{|\text{inv}(v, x_c)|}{l_k}, \tag{5.31}$$

is the invariant distance that the completion state variable must travel in order for the execution to move onto the next group. By definition, for all $v_i, v_j \in V_k$, inv$(v_i, x_c) =$ inv(v_j, x_c). However, with the addition of the Wiener process, $w(t)$, the actual completion rate of the location, $\tilde{l}(v, x_c)$, now is a normally-distributed random variable with mean $l(v, x_c)$. Since the invariant is strictly deterministic in this formulation, the completion time for the group, $\tilde{c}_k$, is also a normally-distributed random variable with mean c_k.

It is possible to approximate the failure probability of a group with this completion time uncertainty to varying degrees of accuracy. For uniform completion groups, the failure probability can be computed for a range of completion times centered about c_k, $C_k^n = \{c^i | c^i \in \mathbb{Z}^+ \wedge c^i \in [c_k - n\sigma, c_k + n\sigma]\}$. Each potential completion time $c^i \in C_k^n$ has an associated adjusted probability of occurring,

$$p(c^i) = \int_{c^i-1}^{c^i} \rho(y)dy - \rho(c^i - 1) \tag{5.32}$$

where ρ is the probability density function of $\tilde{c}_k$. Each potential completion time also has an associated group failure probability, $\tilde{W}_s(k, c^i)$, which is calculated according to equation (5.17).

With this information, one can calculate the estimated group failure probability,

$$\hat{W}_s^n(k) = \sum_{c^i \in C_k^n} p(c^i)\tilde{W}_s(k, c^i). \tag{5.33}$$

This is only a lower bound on the true group failure probability; as $n \to \infty$, $\hat{W}_s^n(k) \to W_s(k)$. However, the lower end of the completion time distribution can be overestimated by taking

$$p(c_{min}^i) = \int_{-\infty}^{c^i} \rho(y)dy \tag{5.34}$$

where

$$c_{min}^i = \min_{c^i \in C_k^n} c^i.$$

The failure probability $\tilde{W}_s(k, c^i)$ increases as c^i increases, so a similar overapproximation is not possible with c_{max}^i.

For the non-uniform completion case, the same process as described for the uniform completion case can be used, however, this assumes that the uncertainty in the different contribution values does not affect the failure paths. For contribution values that are well separated, for example, $\frac{1}{2}$ and $\frac{1}{3}$, this assumption is good for sufficiently small σ. However, as the differences between the contribution

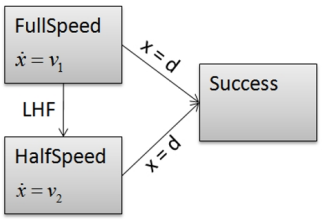


Figure 5.6: Hybrid automaton for the missing state transition example

values get smaller or the uncertainty gets larger, this assumption may no longer apply. In that case, the estimated failure probability will still be a lower bound on the actual failure probability, but increasing n will not make the estimated failure probability converge to the actual failure probability.

5.4.3 Missing State Transitions

The failure probability procedure works for LHA that have discrete transitions based solely on the state of the system. It can be adapted for certain small problems that may have transitions that are also based on the order in which the locations are visited; an example of this is shown in Figure 5.6. This is a similar velocity-controlled rover driving task to the one shown in Figure 5.2, however, once the execution enters into the HalfSpeed location, it must continue there until the task is completed. This extra restriction causes there to be no transition from the HalfSpeed location to the FullSpeed location upon the `LaserHealth` becoming `GOOD`; thus, this is called a missing state transition case.

The complication that arises in missing state transition cases is that assigning each complete system state to just one location becomes impossible. In the example introduced in Figure 5.6, the set of complete system states can be represented by $S = \{\texttt{GG}, \texttt{GF}, \texttt{FG}, \texttt{FF}\}$. While the complete system states with an estimated state of `FAIR` can only occur in the HalfSpeed location, due to the missing transition, the complete system states with estimated values that are `GOOD` are possible in both locations (though they are only possible in the FullSpeed location initially). In order to accommodate this, copies of the ambiguous complete system states must be added to S. If $\text{cloc}(s^\xi, k) = \{v_j | \bigwedge \text{est}(s^\xi, \chi_i) \in \text{ucons}(v_j, \chi_i)\}$ has $n > 1$ members, then each $s^\xi \in S$ must be replaced by n copies identified by location, $\tilde{s}^\xi = \{(s^\xi)^{v_j} | v_j \in \text{cloc}(s^\xi, k)\}$. Then, the adjusted state space becomes $\tilde{S} = S \cup \tilde{s}^\xi$ for all adjusted s^ξ.

In addition to augmenting S, the initial and transition probability vectors and matrices must also

be modified. Let s_i^ξ have an augmented set $\tilde{s}_i^\xi$ and let $v_j \in \text{cloc}(s_i^\xi, k)$ be the location such that $\text{est}(s_i^\xi) \in \text{init}(v_j)$. Then, the initial probability for each $(s_i^\xi)^{v_l} \in \tilde{s}_i^\xi$ is

$$P((s_i^\xi)^{v_l}) = \begin{cases} P(s_i^\xi) & l = j \\ 0 & l \neq j. \end{cases} \tag{5.35}$$

Likewise, the conditional probabilities must be adjusted. Let $e_{jl} \in E$ exist if and only if there exists a valid transition from location v_j to v_l.

Lemma 5.4.1. *For all $v_j, v_l \in cloc(s_i^\xi, k)$, $v_j \neq v_l$, $P((s_i^\xi)^{v_l} | (s_i^\xi)^{v_j}) = 0$.*

Proof. Assume that $\text{est}(s_i^\xi) \models \tau_{jl,k}$ and $v_j \neq v_l$. Since both $v_j, v_l \in \text{cloc}(s_i^\xi, k)$, $s_i^\xi \models \text{inv}(v_j) \wedge s_i^\xi \models \text{inv}(v_l)$. Since the transition scheme is still deterministic and except for the missing transitions, it is state-based, so if the transition does exist, by definition $s_i^\xi \not\models \text{inv}(v_j)$. So, the only location reachable from v_j with a state s_i^ξ is $v_l = v_j$, which negates the starting assumption. □

Lemma 5.4.2. *For each $s_m^\xi \in \tilde{S}$, there exists a unique $(s_i^\xi)^{v_j} \in \tilde{s}_i^\xi$ such that there exists an edge $e_{nj} \in E$ with an associated transition condition $\tau_{nj,k}$ such that est$(s_m^\xi) \models \tau_{nj,k}$, where $v_n \in cloc(s_m^\xi, k)$.*

Proof. If the unadjusted state $s_m^\xi \models \text{inv}(v_j)$, then $v_n = v_j$ and there are no appropriate transition conditions $\tau_{nj,k}$ such that $\text{est}(s_m^\xi) \models \tau_{nj,k}$ by the definition of state-based transitions. If $s_m^\xi \not\models \text{inv}(v_j)$, then there exists some transition edge e_{nj} and condition $\tau_{nj,k}$ that is satisfied by $\text{est}((s_i^\xi)^{v_j})$ and by the definition of state-based transitions, this transition is unique. □

In summary, each complete system state s_i^ξ can transition to only one copy of s_j^ξ with set $\tilde{s}_j^\xi$ because of the overlapping of invariant sets of the locations. In the example, the set $\tilde{S} = \{\texttt{GG}^1, \texttt{GG}^2, \texttt{GF}, \texttt{FG}^1, \texttt{FG}^2, \texttt{FF}\}$. If the original transition matrix for the set $S = \{\texttt{GG}, \texttt{GF}, \texttt{FG}, \texttt{FF}\}$ for the similar example shown in Figure 5.2 looked like this,

$$\mathcal{T} = \begin{bmatrix} 0.5 & 0.05 & 0.05 & 0.4 \\ 0.4 & 0.3 & 0.05 & 0.25 \\ 0.25 & 0.05 & 0.3 & 0.4 \\ 0.4 & 0.05 & 0.05 & 0.5 \end{bmatrix}, \tag{5.36}$$

then the adjusted matrix for set $\tilde{S}$ for the current example looks like this,

$$\tilde{T} = \begin{bmatrix} 0.5 & 0 & 0.05 & 0.05 & 0 & 0.4 \\ 0 & 0.5 & 0.05 & 0 & 0.05 & 0.4 \\ 0 & 0.4 & 0.3 & 0 & 0.05 & 0.25 \\ 0.25 & 0 & 0.05 & 0.3 & 0 & 0.4 \\ 0 & 0.25 & 0.05 & 0 & 0.3 & 0.4 \\ 0 & 0.4 & 0.05 & 0 & 0.05 & 0.5 \end{bmatrix}. \tag{5.37}$$

From this point, the failure probability calculation follows the same procedure for both the uniform and non-uniform completion cases as described previously.

5.5 Problem Complexity and Reduction Techniques

5.5.1 Problem Complexity

The failure probability found using the method described here is exact in the sense that it is not an approximation of or an upper bound on the actual failure probability given the initial information. Although the initial information, like the estimator uncertainty measure and the stationary Markov processes that describe state propagation, are generally approximations, a powerful use for this method is to understand how the failure probability is affected by changes in this initial information.

Unfortunately, the complexity of the failure probability calculation method is exponential in the number of uncertain state variables. Let $y(\chi_i)$ be the number of discrete states in Λ_i for each uncertain state variable $\chi_i \in \mathcal{U}_k$. Then, the number of complete system states is

$$\prod_{\chi_i \in \mathcal{U}_k} y(\chi_i)^2. \tag{5.38}$$

To simplify the problem, let each of the n uncertain state variables, $\chi_i \in \mathcal{U}_k$, $i = 1, ..., n$, have y discrete states in Λ_i. Then, the number of complete system states is y^{2n}. The number of complete system states affects the size of the transition matrices and vectors. The classification of each system state, the calculation of its stationary probability, and the creation of the transition probability matrices and vectors have been automated, however, which allows larger problems to be explored.

Another contributing factor to problem complexity in the non-uniform completion case is the number of distinct contribution values in a group. In general, the number and size of the contribution

values and the completion time affect the number of failure path groups that are possible. The number of failure path groups increases as the number of contribution values and the completion time increase, and decreases as the actual contribution values increase. However, the failure path creation algorithm is very efficient and can handle finding the path groups with little problem. The difficulty then becomes the number of math operations needed to find the failure probability, which is based on the number of complete system states and the number of failure path groups.

5.5.2 Complete System State Reduction Techniques

Because the complexity of the failure probability calculation depends on the number of uncertain state variables and states, techniques to reduce that number are important. One such reduction method is the introduction of derived state variables. A derived state variable is a non-physical state variable whose state propagation completely depends on two or more uncertain state variables. Let $\bar{\chi} \subset \mathcal{U}_k$ be a set of two or more uncertain state variables. Let $\bar{\Lambda}$ be the set of all combinations of discrete states of these state variables,

$$\bar{\Lambda} = \prod_{\chi_i \in \bar{\chi}} \Lambda_i.$$

In some cases, the control of the hybrid automaton may be based on collections of states, $\bar{\lambda}_i \subset \bar{\Lambda}$. If this is the case, a new derived state variable, δ, may be created with discrete sets of states, $\bar{\Lambda}_\delta = \{\bar{\lambda}_1, \bar{\lambda}_2, ..., \bar{\lambda}_m\}$, where

$$m < \prod_{\chi_i \in \bar{\chi}} n_i,$$

the number of individual states in $\bar{\Lambda}$. Therefore the contribution to the problem complexity of the derived state variable would be

$$m^2 < \prod_{\chi_i \in \bar{\chi}} n_i^2. \tag{5.39}$$

An example of this is the `SystemHealth` state variable that is modeled on three sensor health state variables (IMU, GPS, and LADAR), each having two discrete states (`GOOD` and `POOR`). The model of the `SystemHealth` state variable, with three states, and its corresponding hybrid control system is shown in Figure 5.7. In this example, $m = 3$ and $\prod n_i = 8$, so the complexity reduction, $9 < 64$ is significant.

Another way to reduce the complexity of the failure probability calculation is to leverage the state models of composite uncertain state variables. Unlike derived state variables, these uncer-

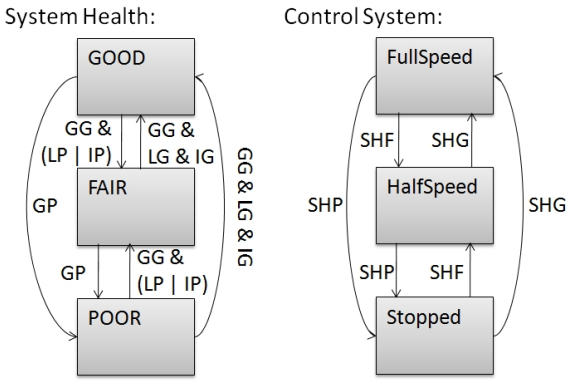


Figure 5.7: Model for derived system health state variable and corresponding hybrid automaton control system

tain state variables are physical and may be adequately described by a stationary Markov process. However, a better model for the state propagation of these state variables may be based on other uncertain state variables. For example, assume that the `LADARHealth` state variable (`LH`) for a robotic system depends on the position of the sun relative to the sensor and the amount of dust on the sensor. This state variable's state propagation can be adequately modeled as a stationary Markov process; however, if the relative position of the sun (`SP`) independently affects the hybrid control system, some reduction in the number of complete system states may be possible. For example, assume that $\texttt{LH} = \chi_1$ has the set of discrete states $\Lambda_1 = \{\texttt{GOOD}^1, \texttt{FAIR}^1, \texttt{POOR}^1\}$ and $\texttt{SP} = \chi_2$ has the set $\Lambda_2 = \{\texttt{DIRECT}^2, \texttt{INDIRECT}^2\}$. Assume also that the model of state propagation of `LH` dictates that if $\text{val}(\chi_2) = \texttt{DIRECT}^2$ then $\text{val}(\chi_1) = \texttt{POOR}^1$. In this case, there can be no complete system state, s, such that

$$(\text{act}(s, \chi_1) \neq \texttt{POOR}^1 \wedge \text{act}(s, \chi_2) = \texttt{DIRECT}^2) \vee (\text{est}(s, \chi_1) \neq \texttt{POOR}^1 \wedge \text{est}(s, \chi_2) = \texttt{DIRECT}^2)$$

is true. This knowledge reduces the total number of complete system states.

5.6 Approximate Methods

5.6.1 Stochastic Hybrid Model Verification

Since methods for verifying certain classes of stochastic hybrid systems exist, it is worth some effort to attempt to construct a suitable stochastic hybrid model for the type of problem solved in this chapter. The hybrid systems treated here assume discrete time execution, so the same assump-

tion will apply to the stochastic hybrid model. The definition for discrete-time switching diffusion processes used in this section is given in Definition 2.3.1.

The hybrid control systems without estimator uncertainty can easily be converted into a type of stochastic hybrid model with probabilistic transitions and deterministic flow equations. The locations, edges, resets, continuous state space and flow equations are one to one between the original hybrid system and the stochastic system; $\mathcal{V}_n = V_o$, $\mathcal{E}_n = E_o$, $X_n = X_o$, and $\phi(X_n, \mathcal{V}_n) = \psi(X_o, V_o)$, where the subscripts stand for "new" and "old," respectively. The transition conditions of the original automaton were based on the discrete states of environment state variables whose state propagation could be modeled by a stationary Markov process. In the conversion, these transition conditions become the transition probabilities between the environment states that satisfy the originating location's invariant and the states that satisfy the accepting location's invariant. For example, let there exist an edge e_{ij} between locations v_i and v_j and let $\Gamma_i = \{s \in S | s \models \text{inv}(v_i)\}$ and $\Gamma_j = \{s \in S | s \models \text{inv}(v_j)\}$. Since the transition condition associated with edge e_{ij} is

$$\tau_{ij} = \bigwedge_{s \in \Gamma_j} s$$

and since for the perfect knowledge case, $\text{act}(s) = \text{est}(s)$, let the transition probability associated with edge e_{ij} be

$$\mu_{ij} = \sum_{s_n \in \Gamma_i} \sum_{s_m \in \Gamma_j} \prod_{\chi_l \in \mathcal{U}_k} P(\text{val}(\chi_l)[\kappa] = \text{act}(s_m, \chi_l) | \text{val}(\chi_l)[\kappa - 1] = \text{act}(s_n, \chi_l)). \tag{5.40}$$

How to add the estimation uncertainty to the stochastic models of the environment variables is not as obvious. Because the hybrid system was verified against the unsafe set in the perfect knowledge case, it is important to distinguish between the actual and estimated states of the system since failure can only occur when these are different. In the uncertain system, the actual and estimated system states have different jobs; the estimated state drives the transitions between the locations of the control system and the actual state in a location can cause the system to reach an unsafe state. In order to differentiate between executing a location nominally and in an unsafe way, a new location, v_u, must be created in each group to account for the unsafe states. So, in the uncertain case, $\mathcal{V}_k = V_k \cup \{v_u\}$.

The transition probabilities between the locations that are not unsafe would also depend on estimation uncertainty in addition to the state propagation probability models. An edge e_{iu} would

be added to each location v_i from which a direct transition into the unsafe set is possible; the transition probability associated with that edge would be the sum of the transition probabilities from each nominal execution state whose estimated state values satisfies the original location's invariant, $s^\xi \models \text{inv}(v_i)$, to each unsafe execution state, s^ω. Let $\Gamma_i^\xi = \{s^\xi \in \Xi_k | \text{ est}(s^\xi) \models \text{inv}(v_i)\}$, then

$$\mu_{ij} = \sum_{s_n^\xi \in \Gamma_i^\xi} \sum_{s_m^\xi \in \Gamma_j^\xi} P(s_m^\xi | s_n^\xi) \tag{5.41}$$

and

$$\mu_{iu} = \sum_{s_n^\xi \in \Gamma_i^\xi} W_{u,k}(n). \tag{5.42}$$

The overall group failure probability would be the probability of reaching the unsafe location. An upper bound of this probability could be found using a variety of existing stochastic hybrid system verification methods.

The problem constructed in this way gives few advantages over the method described previously. The transition probabilities between the locations would need to be calculated for the stochastic hybrid system in much the same way as described previously. Depending on the verification method used, paths through the group may not need to be found explicitly, which may reduce the problem complexity slightly, but the number of complete system states continues to drive the problem complexity. Also, though the number of locations is basically hidden in the previously described failure probability calculation, it is important and even increased slightly in the stochastic hybrid model formulation. Many of the stochastic verification methods available are affected by the number of reachable locations. Finally, most stochastic hybrid system verification methods can only find an approximation of the failure probability, whereas the original method presented here is exact; however, because of the complexity issues with this failure probability calculation, Markov Chain Monte Carlo simulation is a natural next step.

5.6.2 Markov Chain Monte Carlo Simulation

Monte Carlo simulation is a useful way of approximating the failure probability of systems that are too large to reason about using the method described here. Stochastic hybrid systems like those described in the previous section are set up for Monte Carlo simulation; however, getting a system into that form may take more time and/or memory than one has available. The complexity of this problem is exponential in the number of uncertain state variables.

Some systems can be approximated even more. There is an automatic way to enumerate each of the complete system states for a problem and even to sort these states into the appropriate set (Ξ_k, Ω_k, or F_k) for each group V_k, $k = 1, ..., K$. However, calculating the individual transition probabilities for each complete system state can be difficult for large systems. Instead, if the stationary Markov chains modeling the uncertain state variables converge quickly, the equilibrium probability of each complete system state can be automatically calculated and summed over the sets (Ξ_k, Ω_k, and F_k) up to a specified accuracy. For the nominal set, the probability could be broken down further based on the different contribution values.

Let the state propagation models for each uncertain state variable, $\chi_i \in \mathcal{U}_k$, be stationary, ergodic, finite-state Markov processes whose transition matrices, P_i, satisfy

$$P_i = \lim_{n\to\infty} P_i^n. \tag{5.43}$$

Let the estimation uncertainty matrix, $E_{\chi,i}$, for each uncertain state variable be symmetric; also, let the probability of estimating the correct state be the same for each possible state. Finally, let the augmented probability matrix for each state variable, $P_{\chi,i}$, be the matrix that gives the transition probability between complete states of the individual uncertain state variable. By abuse of notation, let $s \in \Lambda_i \times \Lambda_i$ be a complete state of single uncertain state variable χ_i, and let $\text{act}(s) \in \Lambda_i$ and $\text{est}(s) \in \Lambda_i$. For any two states $s_j, s_l \in \Lambda_i \times \Lambda_i$, the augmented transition probability is

$$\begin{aligned} P_{\chi,i}(j,l) = P(\text{val}(\chi_i)[\kappa] = \text{act}(s_l) | \text{val}(\chi_i)[\kappa-1] = \text{act}(s_j)) \times \\ P(\text{val}(\hat{\chi}_i)[\kappa] = \text{est}(s_l) | \text{val}(\chi_i)[\kappa] = \text{act}(s_l)). \end{aligned} \tag{5.44}$$

Proposition 5.6.1. *The augmented transition probability matrix, $P_{\chi,i}$, satisfies*

$$P_{\chi,i} = \lim_{n\to\infty} P_{\chi,i}^n \tag{5.45}$$

if the original transition probability matrix P_i also satisfies the same equation.

Proof. Because P_i satisfies Eq. (5.43), $n_i \times n_i$ matrix is a column vector n_i $1 \times n_i$ vectors, π,

$$P_i = \begin{bmatrix} \pi \\ \vdots \\ \pi \end{bmatrix}, \tag{5.46}$$

where π is the stationary distribution of P_i,

$$\pi = \pi P_i. \tag{5.47}$$

Therefore, the jth column of P_i is a $n_i \times 1$ vector of value $\pi(j)$. Since the estimation probability is based on the actual value of the state variable at time κ instead of at time $\kappa - 1$, the augmented matrix is created as follows. The jth column of $P_{\chi,i}$ corresponding to a state s such that act$(s) = \lambda_j$ and est$(s) = \lambda_l$ is

$$P_{\chi,i}(j) = \vec{\pi}(j) P(\text{val}(\hat{\chi}_i) = \lambda_l | \text{val}(\chi_i) = \lambda_j) \tag{5.48}$$

where $\vec{\pi}(j)$ is a column vector of n_i $\pi(j)$ values. Since the constant row vector is multiplied by a constant, the resulting column vector of $P_{\chi,i}$ is also a constant vector. This is true for all columns of $P_{\chi,i}$; therefore,

$$P_{\chi,i} = \begin{bmatrix} \pi_\chi \\ \vdots \\ \pi_\chi \end{bmatrix}, \tag{5.49}$$

where π_χ is the stationary distribution of the augmented transition probability matrix and $P_{\chi,i}$ satisfies Eq. (5.45). □

Proposition 5.6.2. *The composition of augmented matrices, $\tilde{P}_\chi = P_{\chi,1} \circ P_{\chi,2} \circ ... \circ P_{\chi,N_k}$, satisfies*

$$\tilde{P}_\chi = \lim_{n \to \infty} \tilde{P}_\chi^n. \tag{5.50}$$

Proof. The proof is by construction and is similar to the proof of Proposition 5.6.1. Since the column vectors of each of the augmented matrices are constant vectors, the column vectors of $\tilde{P}_\chi$ are also constant vectors. Therefore,

$$\tilde{P}_\chi = \begin{bmatrix} \tilde{\pi} \\ \vdots \\ \tilde{\pi} \end{bmatrix} \tag{5.51}$$

where $\tilde{\pi}$ is the stationary distribution of $\tilde{P}_\chi$ and $\tilde{P}_\chi$ satisfies Eq. (5.50). □

These two well-known results show that manipulating the stationary Markov chain in the given ways does not change its desired properties. By Proposition 5.6.2, the Markov chain that controls the transitions between complete system states has reached its stationary or equilibrium distribution

initially if each uncertain state variable's stationary Markov chain modeling its state propagation also starts out at the stationary distribution. Thus, the mixing time (time to reach the stationary distribution within a given error) is zero and the stationary probabilities can safely be used in the failure probability estimation. There is an automatic algorithm that can find each complete system state, calculate its stationary probability, and place it into the appropriate set (Ξ_k, Ω_k, and F_k). Then, the stationary probability of each set can be calculated from the sum of the stationary probabilities of its elements. Since there may be many complete system states with a negligible stationary probability, the algorithm sorts the elements so that those with greater probability values are placed into the sets first, and once the sum of the set probabilities reaches a pre-determined value, the algorithm aborts. The pre-determined value must be chosen so that the remaining probability can be assigned to the unsafe set without too much conservatism.

In a uniform completion case, the paths could easily be found and an approximation of the failure probability calculated. For more complicated examples, including non-uniform completion cases and cases with uncertain completion times, Markov Chain Monte Carlo simulation is a useful way to find the failure probability approximation.

5.7 Conclusion

A formal method for calculating the probability of a verifiable sensor-driven hybrid system entering into a specified unsafe set due to estimation uncertainty was presented. The calculation of the failure probability of this system gives the designer some information about the control system. If the failure probability of a given system is too high for the design requirements, several changes could be made. First, the estimator for the state variable could be improved; for some cases, a better sensor could be used to reduce the probability of failure; and finally, the control system could be designed to depend less on a relatively unknown state variable. The verification of control systems in the presence of different forms of uncertainty, including estimation uncertainty, is an important problem, and this approach seems promising as a design tool for hybrid control systems with state-based transitions. These techniques have been applied on two significant examples which are described in the next chapter.

Chapter 6

Significant Goal Network Verification Examples

6.1 Introduction

The three verification methods introduced so far are applied to two examples in this chapter. The first example is a goal network for a complex rover system that is based on autonomous robotic systems such as the DARPA Grand (or Urban) Challenge vehicles. This example has twelve state variables and twenty-one controlled goals, so is not too large for the conversion procedure and PHAVer verification method. It is difficult to apply the failure probability calculation method without some model reduction techniques, however. This example is described in Section 6.2. The second example, described in Section 6.3, is a goal network for an example mission to Titan, a moon of Saturn. The autonomous aerobot probe must explore the lower atmosphere of Titan and map its surface while staying safe and performing other tasks. This example is sizable; over 500 locations are found using the conversion procedure. The number of passively constrained state variables after some model reduction techniques is nine, but that proves to be too much for the conversion procedure because of the number of failure transitions. The conversion procedure is able to convert the goal network after applying a restricting assumption, but even then, PHAVer is not able to verify the system (though resorting to overapproximation and other abstraction techniques may have helped). So, this example is redesigned and verified using the novel verification method described in Chapter 4. The failure probability calculation must be approximated for this example following the Markov Chain Monte Carlo technique outlined in Chapter 5.

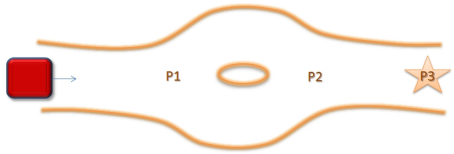


Figure 6.1: A depiction of the example task where the rover must traverse a path and reach the end point, which is marked with a star

6.2 Complex Rover Example

This example is based on a possible set of commands for autonomous rovers such as Mars exploration robots or DARPA Grand Challenge vehicles. The problem size is small but the system has enough complexity to begin to test the capabilities of the conversion software and the verification method.

6.2.1 Goal Network Design

This example involves an autonomous rover whose ultimate goal is to follow a given path to a specified end point, shown in Figure 6.1. This rover has three main sensor systems: GPS, LADAR, and an inertial measurement unit (IMU). The path choice and speed limit along the chosen path is dependent on the combined health of these sensors. Each sensor degrades or fails in a specified way. The GPS can experience periods of reduced accuracy (satellite dropouts) or failure (electrical or structural signal interference), and these can both be modeled as recoverable stochastic events. The health of the LADAR depends on the location of the sun in the sky. If the sun is shining directly into the LADAR, its measurements cannot be used. Some degradation of the LADAR's capabilities occur at less direct sun angles, as well. Finally, the health of the IMU is dependent on the temperature of the device. If the temperature of the IMU is too low, a heater can be used to heat the sensor. If the IMU temperature gets too high, the unit must be powered off. The state effects diagram listing all the state variables important to the system is shown in Figure 6.2. State variables, measurements, and commands are shown and the arrows between these indicate modeled effects on the accepting state variables or measurements.

The state variable types for each state variable can be found in Table 6.1. The goal network for this task is shown in Figure 6.3 and the individual goal trees are shown in Figures 6.4–6.6. The first goal tree describes the path the robot will take, the second constrains the speed limits that will apply to the robot, and the third describes the IMU temperature management method.

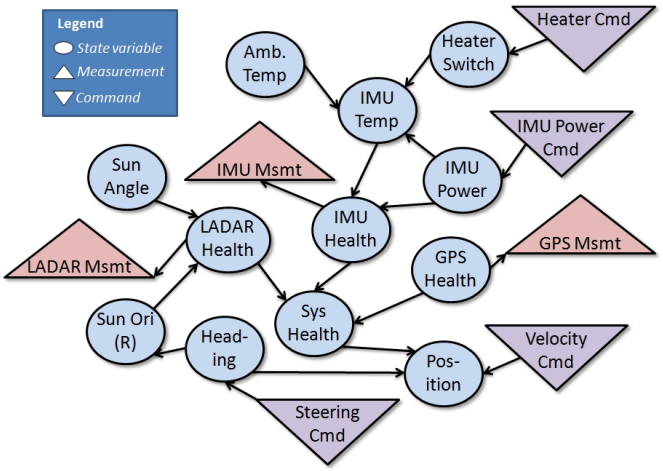


Figure 6.2: State effects diagram for the complex rover example

Table 6.1: State Variable Data

State Variable	Abbreviation	Type
Position	x	Controllable
Heading	θ	Controllable
IMU Power	ps	Controllable
Heater Switch	hs	Controllable
IMU Health	IH	Dependent
Rel. Sun Orientation	SO	Dependent
LADAR Health	LH	Dependent
System Health	SH	Dependent
IMU Temperature	IT	Dependent
GPS Health	GH	Uncontrollable
Ambient Temp.	AT	Uncontrollable
Sun Angle	SA	Uncontrollable

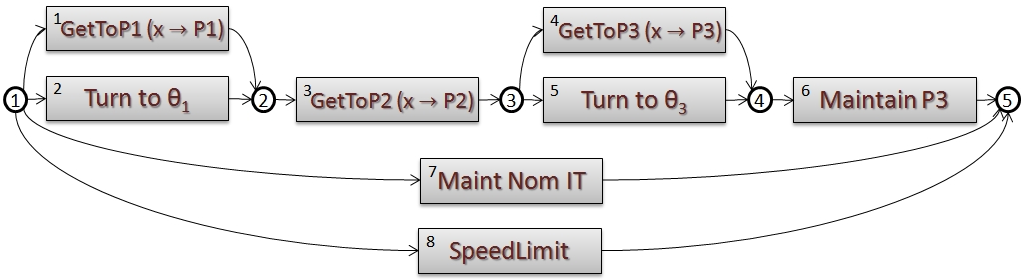


Figure 6.3: Goal network for the complex rover example

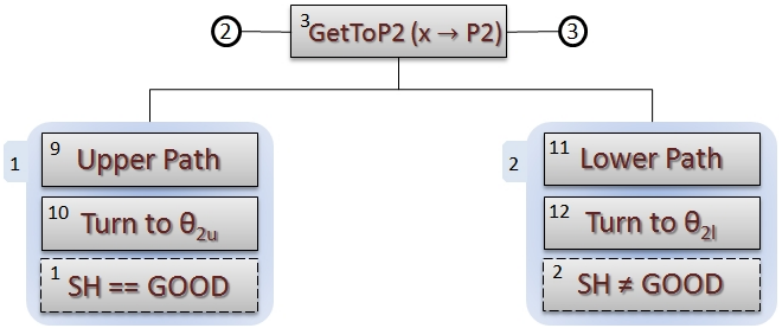


Figure 6.4: Path goal tree

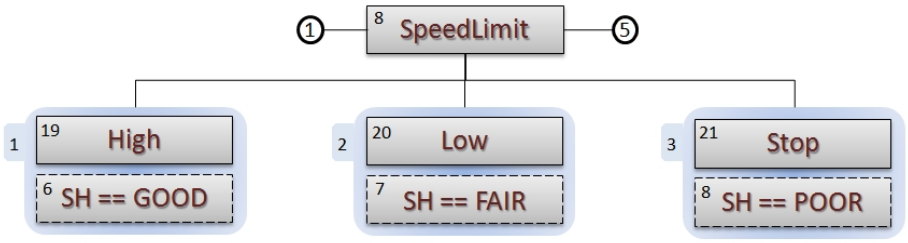


Figure 6.5: Speed limit goal tree

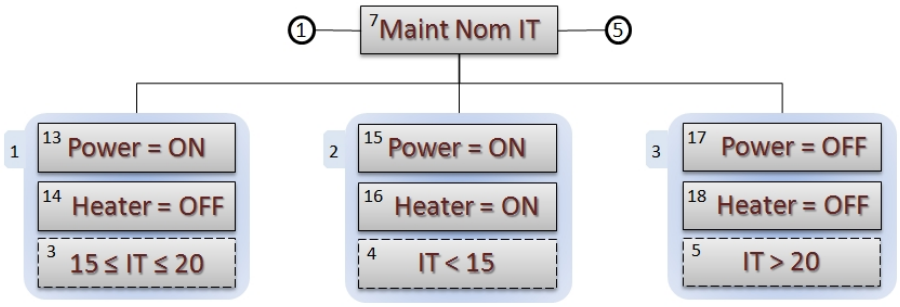


Figure 6.6: IMU Temperature goal tree

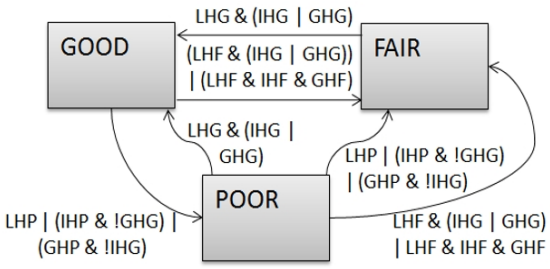


Figure 6.7: A finite state machine that describes the model of the `SystemHealth` derived state variable. IHG = `IMUHealth` is `GOOD`, GHF = `GPSHealth` is `FAIR`, LHP = `LADARHealth` is `POOR`, etc.

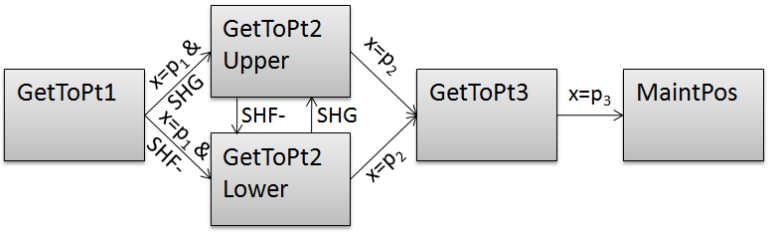


Figure 6.8: The path automaton. One of three automata that are composed into the control system for the rover. These four sets of locations represent groups 1–4.

6.2.2 Conversion and Verification

The goal network control program for this example consists of twenty-one controlled goals, including eight root goals, and twelve state variables. The conversion software found thirty-eight locations (including the Success location) in four groups in the goals automaton. Figures 6.8 and 6.9 show the hybrid systems whose composition creates the goals automaton. In addition to this, there are eight other automata that describe the state models of the dependent and uncontrollable state variables. One of these, the `SystemHealth` state variable model, is shown in Figure 6.7. In all, the composition of the nine automata creates a discrete state space with over 200,000 states. The unsafe set for this problem consists of the following conditions:

1. The rover is not stopped ($\dot{x} \neq 0$) when the IMU is off (`ps` $==$ `OFF`) and the GPS is degraded (`GH` $\neq$ `GOOD`).

2. The rover moves forward ($\dot{x} \neq 0$) when the sun is pointing directly into the LADAR unit (`LH` $==$ `POOR`).

While the conversion software took less than five seconds to generate the PHAVer code for this system, PHAVer was not able to handle the large state space without resorting to overapproximation.

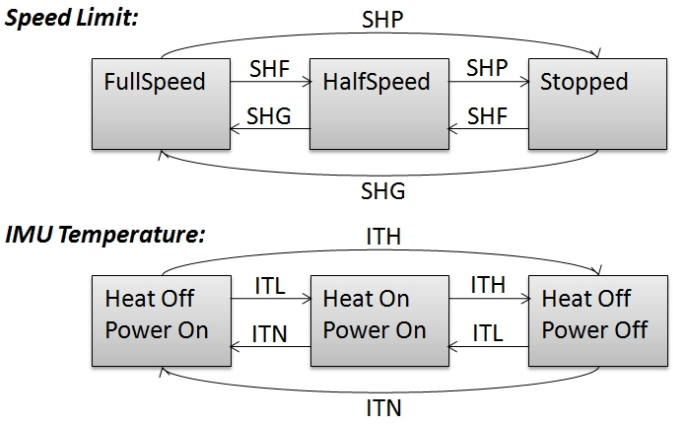


Figure 6.9: The speed limit and IMU temperature automata; all locations in the two automata span groups 1–3

So, several reduction techniques were employed. The first reductions were in the uncontrollable and dependent state variable automata. The `SunAngle` and `RelativeSunOrientation` state variables, which were modeled with stochastic transitions, were removed and the `LADARHealth` state variable's model became stochastic. Since these two state variables were not used elsewhere in the goal network, this simplification did not affect the quality of the model. Next, the `AmbientTemperature` state variable was removed as was the `IMUTemperature` state variable's dependence on it, simplifying the model. In the last model reduction, the `SystemHealth` state variable was removed in favor of using the three sensor health state variables in its place. These reductions made the discrete state space a more manageable 3726 states. The final reduction was to verify the goals automaton group by group, which is possible because the unsafe set did not constrain the progress of the `Position` or `Orientation` state variables, though the initial condition problem must be handled carefully.

The goals automaton could be verified after making some corrections. The verification software found reachable states in the automata that entered the unsafe set, so the goals automaton had to be corrected to ensure that the unsafe set was not entered. The transitions into the locations where the `IMUPower` is off (`ip` $==$ `OFF`) and the speed is not zero ($\dot{x} \neq 0$) must also have a condition that the `GPSHealth` is `GOOD` or `FAIR` (`GH` $\neq$ `POOR`)) to satisfy the unsafe set. These changes were added, verified, and then translated back into the goal network by adding a new tactic in the speed limit goal tree, which can be found in Figure 6.10. This makes the control program conservative (more states than necessary are constrained to have zero rate) but verifiable with respect to the given unsafe set.

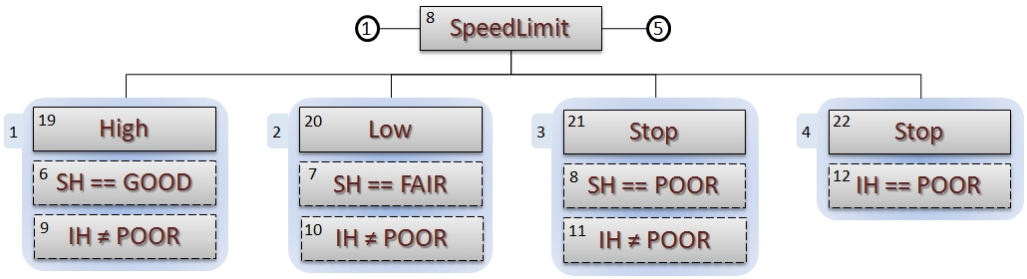


Figure 6.10: Redesigned speed limit goal tree

6.2.3 Uncertainty Analysis

The goal network verified in the previous section can now be analyzed for safety in the presence of state estimation uncertainty. The unsafe set specified in the previous section continues to be the set of conditions that the uncertainty analysis will use. Since the first three groups, V_1, V_2, and V_3, are essentially the same set of locations repeated, only V_1 will be analyzed. The velocity in V_4 is constrained to be zero, so there is no way to enter the unsafe set based on the uncertain state variables; the failure probability for this group is $W_s(4) = 0$. The system has four uncertain state variables (`IMUTemperature`, `GPSHealth`, `LADARHealth`, and `IMUHealth`), each with three possible state values ($\{$`GOOD`, `FAIR`, `POOR`$\}$ for the health state variables and $\{$`LOW`, `NOMINAL`, `HIGH`$\}$ for `IMUTemperature`). Since that translates into $3^8 = 6561$ complete system states, simplification is necessary. Instead, if the `SystemHealth` state variable with three states replaces the two of the sensor health state variables and a two state `IMUHealth` is used ($\{$`POOR`, `NOTPOOR`$\}$), the number of complete system states reduces to $2^2 * 3^4 = 324$.

Another observation is that the `IMUHealth` depends on the `IMUTemperature`. Since there is a model that controls what the `IMUHealth` is estimated to be given the `IMUTemperature`, the estimated `IMUHealth` is known given the `IMUTemperature`. However, the actual `IMUHealth` may not always be known given the actual `IMUTemperature` due to modeling errors. In certain cases, such as when the estimated `IMUTemperature` causes the `IMUPower` to be turned OFF, that the actual `IMUHealth` state is known given the estimated `IMUTemperature`. This dependence of an uncertain state variable on another causes the number of complete system states to be further reduced. Dependencies between two state variables is handled by creating a new composition state variable that consists of all the possible actual and estimated states that the two variables can take given the dependencies. For this problem, the new state variable is called `TI` and has 18 states, which are made up of estimated and actual values of `IMUTemperature` and `IMUHealth`.

With the nine possible states for the estimated and actual values of the `SystemHealth` state variable, the new total number of complete system states is $18 * 9 = 162$.

The group V_1 is a non-uniform completion group with three different contribution values, 1, $\frac{1}{2}$, and 0, that correspond to the Full Speed, Half Speed, and Stopped speed limit tactics, respectively. The completion time for V_1 is assumed to be $c_1 = 5$. Based on the unsafe set specification and the composed hybrid automaton control system, the nominal set, Ξ_1, has 120 complete system states, the Safing set, F_1, is empty, and the unsafe set, Ω_1 contains the remaining 42 complete system states. The state transitions of the two remaining uncertain state variables are modeled as stationary Markov processes; in this case the models were chosen to have what was considered to be realistic values, but in practice, these models would be chosen based on simulations or tests of the hardware. For this example, several different estimator uncertainty values were chosen and the failure probability was calculated for each. Using these models, uncertainty values, and the sets of complete system states, the appropriate vectors and matrices were calculated. The initial failure probability, a_1, is the sum of the initial probabilities of all 42 unsafe complete system states. There are three initial probability vectors W_1^i corresponding to the three contribution values; the dimensions of W_1^1 is 1×6, W_1^2 is 1×12 and W_1^3 is 1×102. There are nine Q_1 matrices between the β groups and three $W_{u,1}$ vectors, with the same dimensions as the W_1 vectors.

Since this case has a set of locations that has zero contribution value, there are an infinite number of failure paths. However, by using the power series equation,

$$\sum_{x=0}^{\infty} Q^x = (I - Q)^{-1}, \tag{6.1}$$

each failure path can be accounted for in the failure probability calculation. The failure probability was calculated for several values of estimation uncertainty and the results are shown in Fig. 6.11.

6.3 Titan Aerobot Example Mission

Titan is the largest moon of Saturn and is remarkable for its dense atmosphere that has an estimated composition of 95% nitrogen, 3% methane, and 2% argon. The surface pressure on Titan is about 1.5 bars, which is 1.5 times the surface pressure on Earth. The thick atmosphere and the methane haze make surface observation difficult; however, near infrared observations and pictures from the Huygens probe suggest that interesting terrain is present and is made of solid rock and frozen water

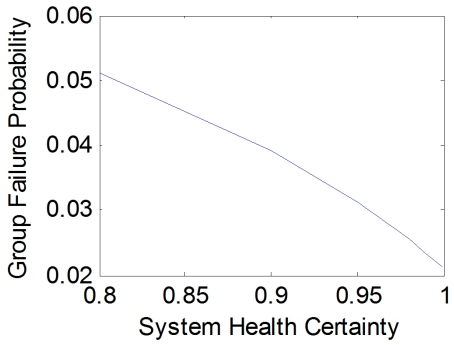


Figure 6.11: Group failure probability vs. SH estimation certainty

ice littered with liquid methane and ethane bodies. Cryovolcanism has been conjectured, as has a methane and ethane cycle like the water cycle present on Earth [82].

A proposed mission to Titan consists of a satellite of Titan that would release an aerobot probe to the lower atmosphere. This lighter-than-air vehicle would use wind currents to explore the moon by taking advantage of Titan's unique atmosphere. The probe would have the capability to fly to points while simultaneously mapping Titan's surface; it would also be able to stationkeep. In addition to wind profiling, surface and atmospheric observations, and atmospheric composition testing, the probe would also have the capability to collect samples from the surface without landing [1].

Because the Saturn system is far away from Earth, there is a significant light-time delay of about 2.6 hours round-trip [82]. This means long communication latencies between the aerobot and Earth. Having an autonomous vehicle that can execute a relatively long mission plan without human interference is important. This autonomous control system must be able to function without human intervention and be able to identify and handle many types of faults and failures in a safe manner. The verification of the fault-tolerant control plans against sets of unsafe conditions will be extremely important and useful for a Titan exploration mission.

6.3.1 Problem Statement

A simplified model of the Titan aerobot was used as an example for the conversion and verification procedure. The aerobot used in this example has a mission to fly to a specific area while maintaining at least 10% power, position awareness, and appropriate safe altitudes, and while being aware of spontaneous science observation opportunities. The example aerobot has several sensors, including two cameras, a laser range finder (LRF), a radar, a hygrometer, and a motion sensor. The cameras and laser range finder allow the aerobot to localize and map the surface of Titan while maintaining

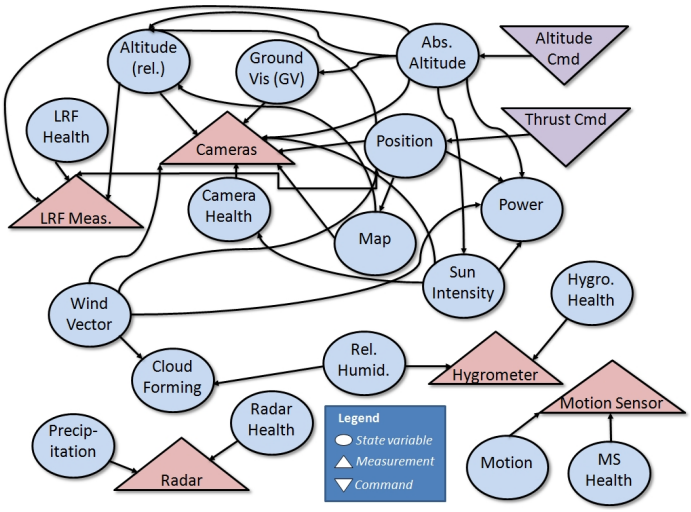


Figure 6.12: State effects diagram for the aerobot example

a safe altitude above Titan's surface features. The radar, hygrometer, and motion sensor are used to detect spontaneous science events such as cloud formation, precipitation, and cryovolcanism. Figure 6.12 gives the state effects diagram for this example problem. The state effects diagram lists all pertinent state variables, commands, and measurements that are used to control the system. The arrows between the different bodies in the diagram indicate that the originating body has a modeled effect on the accepting body.

Most of the models between state variables or between state variables and measurements or commands are fairly obvious. The aerobot is able to localize using the existing map, which is generated by the orbiting satellite and to which details are added by the aerobot. The map uncertainty is a measure of the scale of the surface feature information in an area; the uncertainty is high in areas only covered by satellite images and is low in areas imaged by the aerobot. The position uncertainty, then, is a measure of how well the aerobot can constrain its position relative to the existing map. Sunlight intensity and ground visibility are affected by the aerobot's absolute altitude, and these state variables affect the quality of the measurements taken by the cameras. Relative altitude is the height that the aerobot is above the ground and is the state that the LRF is measuring. The aerobot's position is controlled by the thrust and altitude commands and affected by the wind vector. Power is also affected by the wind vector because more or less control effort may be needed to drive the aerobot based on the direction and magnitude of the wind. The aerobot is assumed to have some regenerative power capability based on solar energy, so sunlight intensity also affects the percentage of power remaining on the probe. (However, with Titan located so far from the sun, solar energy

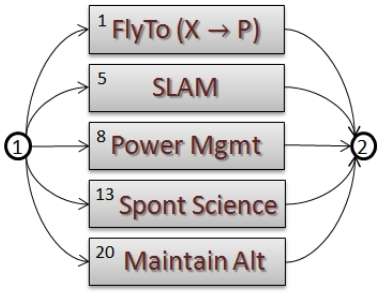


Figure 6.13: Goal network for Titan example

could at best be a back-up power system for an actual mission.) It is also assumed that altitude affects sunlight intensity, with more intensity near the top of the atmosphere.

A derived state variable is a non-physical state variable that depends only on other state variables. The `SpontaneousScience` state variable with four states (`NONE`, `MOTION`, `PRECIP`, `CF`) is a derived state variable that depends on seven state variables (`Motion`, `Precipitation`, `CloudForming`, `RelativeHumidity` and the accompanying sensor health state variables). The `SpontaneousScience` state variable prefers the rarer events; if motion is sensed, then that is the value of the state variable regardless of the presence of precipitation or cloud formation. The next preferred event is precipitation followed by cloud forming.

6.3.2 Goal Networks

The goal network for this example problem, shown in Figure 6.13, is based on the control of the position and altitude of the aerobot as it flies to a specified point. The `Position` state variable (`X`) is controlled via three modes: a "fly to" mode where the aerobot heads towards the constrained area; a "stationkeeping" mode where the aerobot maintains its current position; and a "float" mode where the aerobot drifts without controlling its position. There are also five control modes for the `Altitude` state variable (`Z`) that refer to different absolute altitudes; from lowest to highest, these altitudes are ground observation, detailed mapping, minimum en route, maximum terrain clearance, and service ceiling. The abbreviations used for the passive state variables can be found in Table 6.2.

The first concurrently executed goal tree, shown in Figure 6.14, involves the task of flying to a specified area. There are two tactics available for doing this that are chosen based on the relative wind vector. When the wind vector is favorable or small, the aerobot attempts to maintain a minimum velocity in the direction of the specified area. When the wind vector is large and unfavorable,

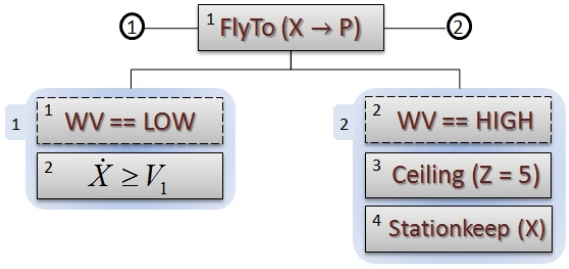


Figure 6.14: Goal tree for flying to a specified area

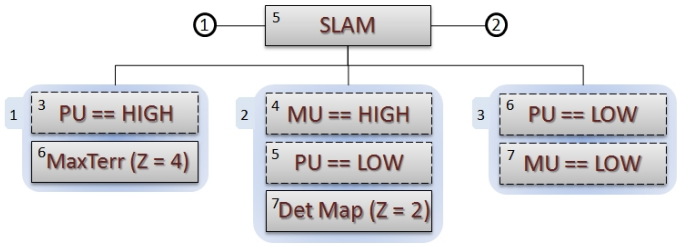


Figure 6.15: Goal tree for simultaneous localization and mapping

the aerobot instead attempts to stationkeep; in a more complex example, the aerobot would profile the wind to find a new altitude at which to fly.

How well the aerobot can constrain its position on the existing map contributes to the position uncertainty; when the uncertainty is high, the aerobot must ensure that it is at a safe altitude to avoid controlled flight into terrain. The second goal tree, shown in Figure 6.15, gives tactics that help to accomplish the task of simultaneous localization and mapping (SLAM). When position uncertainty is high, the aerobot ascends so as to clear all possible obstacles and to match its position with the less detailed satellite map. Execution continues as usual when the position and map uncertainty are low. If the position uncertainty is low and the map uncertainty is high, the aerobot flies at a lower altitude to achieve more detailed mapping.

The third task for the aerobot is power management, which is controlled in the goal tree shown in Figure 6.16. Overall, the aerobot must maintain at least 10% power; if it does not, the aerobot safes to floating until the power increases. While the power value is above 10%, there are several tactics to ensure that the power level does not drop to the safing level. When the power drops below 50%, the aerobot climbs to increase the sunlight intensity that it is receiving. If the power dips below 30%, the aerobot discontinues its trek to the specified point and instead stationkeeps to preserve power.

Spontaneous science observation is an important part of the Titan aerobot's mission, so the goal

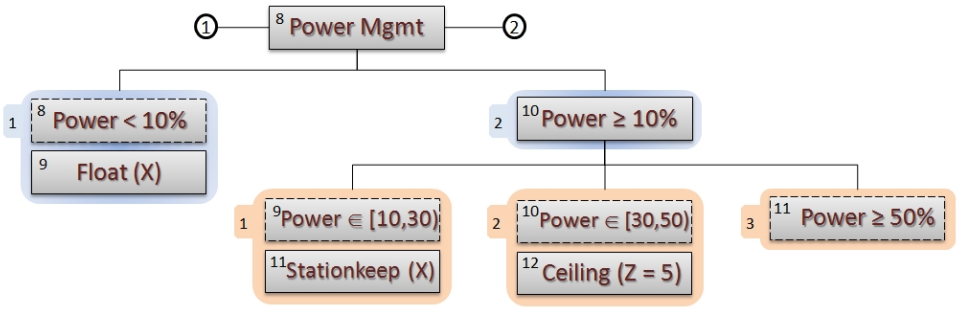


Figure 6.16: Goal tree for power management

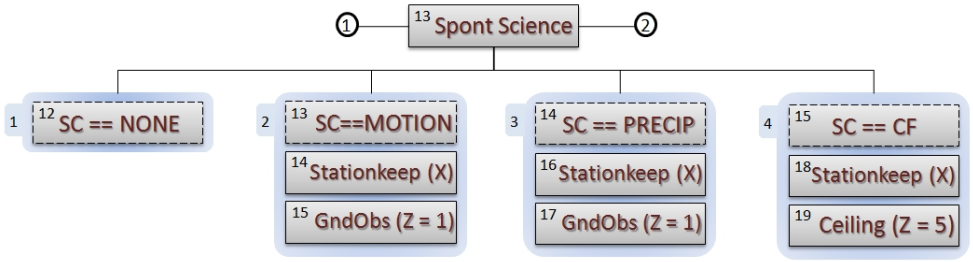


Figure 6.17: Goal tree for observing spontaneous science

tree shown in Figure 6.17 deals with this task. When no motion, precipitation, or cloud formation is detected, the aerobot continues on with its current task. However, if motion on the surface (such as cryovolcanism) or precipitation is detected, the aerobot descends and stationkeeps to observe it. Likewise, if cloud formation is detected, the aerobot ascends to observe it.

Several other factors also affect the altitude at which the aerobot flies, such as the health of the position sensors (the cameras and the LRF), the ground visibility, and sunlight intensity. These conditions make up the five tactics of the final goal tree controlling the altitude of the aerobot; this is shown in Figure 6.18.

Each of the goal trees presented are executed concurrently in the aerobot's goal network. It is assumed that when the aerobot has positive control (i.e., the position and map uncertainties are low, the wind vector is low, etc.), the low-level position controllers successfully avoid terrain at the low altitudes. When two goals constraining altitude are merged (executed concurrently), the higher altitude is taken; likewise, when two goals constraining position are merged, float constraints take priority, and then stationkeeping constraints are preferred over fly to constraints.

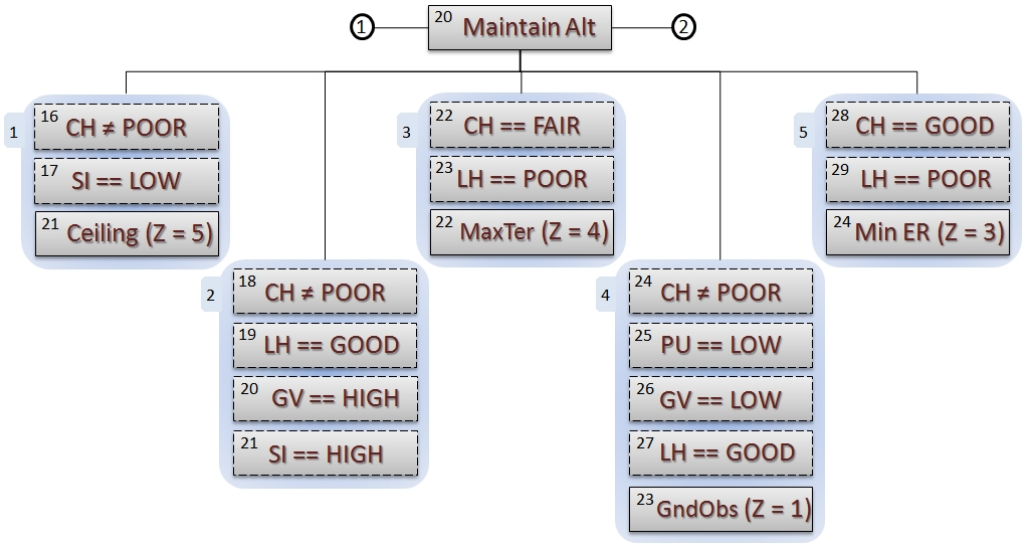


Figure 6.18: Goal tree for controlling the altitude of the aerobot

6.3.3 Verification

The control system for the Titan aerobot mission designed in Section 6.3.2 was converted to a hybrid automaton with 544 locations and thousands of transitions (using a restrictive simplifying assumption that only a single passive state can change per time step) and an input file to the PHAVer symbolic model checker was automatically generated. The unsafe set for the goal network, $Z = \{\zeta_1, \zeta_2\}$, had two sets of constraints:

1. Power is less than 10%, $\zeta_1 = \{(\texttt{Power}, <, 10)\}$.
2. The altitude is lower than maximum terrain clearance while the ground visibility is low and the position uncertainty is high, $\zeta_2 = \{(\texttt{z}, <, 4), (\texttt{GV}, ==, \texttt{LOW}), (\texttt{PU}, ==, \texttt{HIGH})\}$.

PHAVer was not able to handle the automaton along with the nine passive state variable model automata due to the large state space that results (over 2.5 million discrete states). The list of passive state variables and the number of discrete states in the model of each are given in Table 6.2.

To handle the large verification effort, the method introduced in Chapter 4 was used. The first step of that procedure is to ensure that the goal network has state-based transitions. Each root goal and its goal tree were run through the SBT Checker and the software found that the altitude controlling goal tree was missing a tactic with passive constraints as follows:

$$\texttt{LH} == \texttt{GOOD} \wedge \texttt{SI} == \texttt{HIGH} \wedge \texttt{GV} == \texttt{LOW} \wedge \texttt{PU} == \texttt{HIGH} \wedge \texttt{CH} \neq \texttt{POOR}.$$

Table 6.2: Passively Constrained State Variables

State Variable	Abbreviation	Number of States	Estimator Accuracy
Camera Health	CH	3	0.95
LRF Health	LH	2	0.95
Sun Intensity	SI	2	0.99
Ground Visibility	GV	2	0.99
Wind Vector	WV	2	0.99
Position Uncertainty	PU	2	0.9
Map Uncertainty	MU	2	0.9
Spontaneous Science	SC	4	0.8
Power		6	

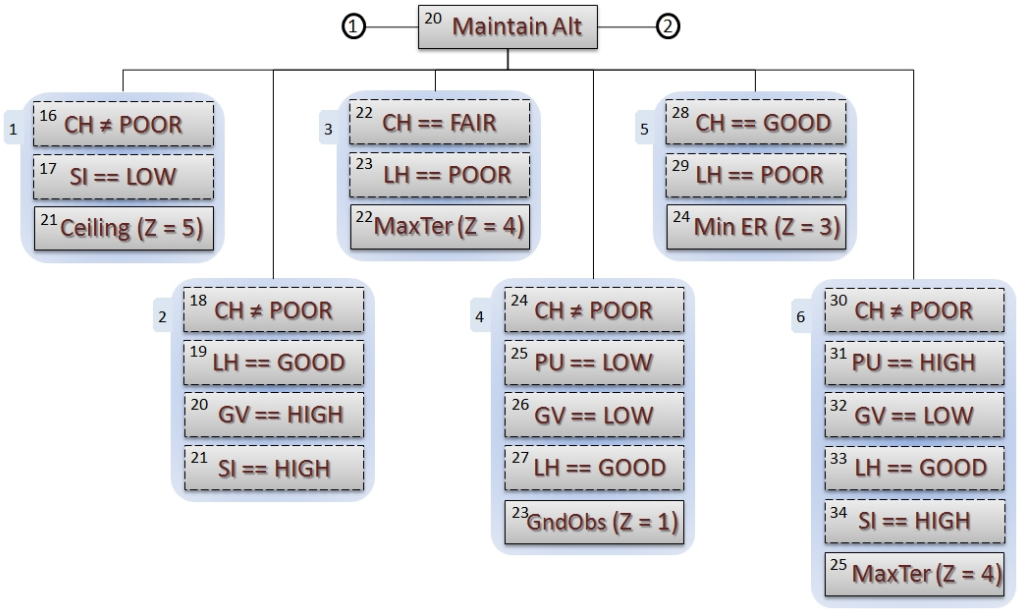


Figure 6.19: Redesigned altitude control goal tree

The unsafe set, ζ_2, dictates that the altitude must be constrained in this tactic to be either the maximum terrain clearance altitude or the service ceiling ($\mathtt{z} = 4$ or $\mathtt{z} = 5$, respectively). Since the sun intensity is high, the maximum terrain clearance altitude is the appropriate constraint. The new tactic in the redesigned altitude control goal tree is shown in Figure 6.19.

The goal network was then verified using the InVeriant software. More than 600 locations were generated and no inconsistent controlled constraints were found. The verifier composed the converted automaton with the `Power` state variable's model automaton for the first unsafe set, ζ_1. The verifier found locations in which the unsafe power constraint was possible when $10 \leq \mathtt{Power} < 30 \wedge \mathtt{WV} == \mathtt{HIGH} \wedge \mathtt{SI} == \mathtt{LOW}$. In order to most efficiently use the software, the unsafe power constraint, ζ_1, was converted into two equivalent constraints: $10 \leq \mathtt{Power} < 30 \wedge \frac{d}{dt}(\mathtt{Power}) > 2$.

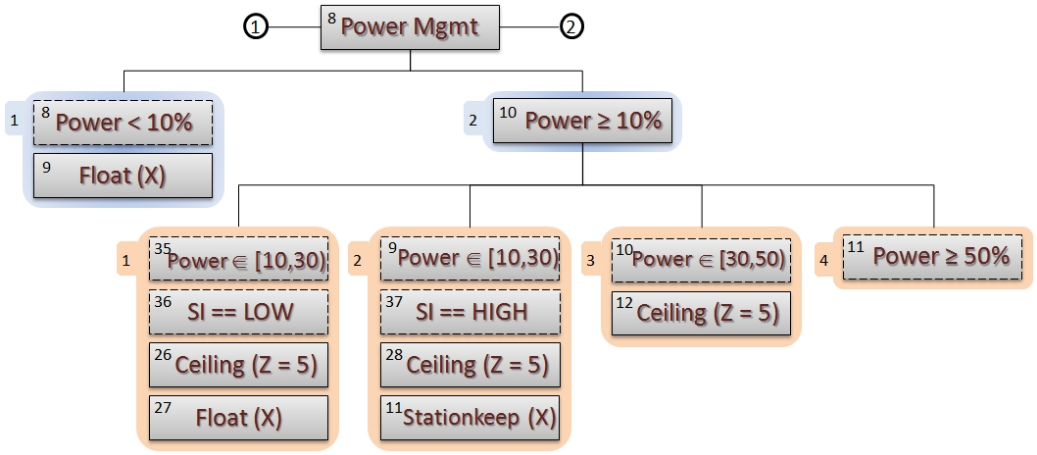


Figure 6.20: Redesigned power management goal tree

The constraint on the power rate is equivalent to a medium or high power use state; if these power rates are present in a location where the power falls into the given constraint, it is possible to achieve a power state that is less than 10%. Because the `Power` state variable is a continuous, rate-driven dependent state variable, a path from the initial condition ($\texttt{Power} = 100$) to the unsafe condition ($\texttt{Power} < 30$) must be found to prove that the unsafe locations are reachable. The software was able to find a path of three locations whose invariants included the appropriate power constraints ($\texttt{Power} \geq 50$, $30 \leq \texttt{Power} < 50$, and $10 \leq \texttt{Power} < 30$) and whose power rates were negative, proving that the unsafe conditions were reachable. The software also output the goals that were common to all the unsafe locations, which triggered the redesign of the power management goal tree to include a tactic that commands the aerobot to float at the service ceiling when the `Power` is less than 30% and the `SunIntensity` is `LOW`. This new power management goal tree is shown in Figure 6.20. Verification of the redesigned goal network confirmed its correctness.

The SBT checker and InVeriant verification software package is superior to the conversion software and PHAVer for this application. The conversion software was able to handle the large goal network to linear hybrid automaton conversion with the transition restriction, taking just under five hours on a 2.0 GHz Intel Core 2 Duo CPU with 4.0 GB of RAM. However, many verification attempts using PHAVer proved to be unfruitful. While this does not show that PHAVer could not complete the task, abstraction, model reduction, and overapproximation would be necessary. The SBT Checker, however, took nearly no time to run once the appropriate data were entered and the output of that software was useful to the design process, unlike the output of the conversion procedure. Then, the InVeriant software was able to convert the goal network into locations and

invariants in about fifteen minutes, followed by about two minutes of verification work. Whereas PHAVer outputs conditions on the state variables that allow at least one of the unsafe conditions to be true, InVeriant gives that information along with the unsafe constraint satisfied and the goals and tactics that are responsible for the failure.

6.3.4 Uncertainty Analysis

The uncertainty analysis was completed for the Titan aerobot mission. The large number of potentially uncertain state variables caused an explosion in the number of complete system states. Assuming that all state variables in Table 6.2 except Power are uncertain, the number of complete system states was almost 600,000. Only the second unsafe set condition was analyzed for simplicity.

The failure probability of the second unsafe set condition was calculated using the automatic complete system state sorting software. The fourth column of Table 6.2 gives the probability that the state variables' estimators are correct. Stationary Markov chains with no mixing time were assumed for the state propagation models for each of the uncertain state variables. After about 65 hours of computation on the computer described above, a stationary unsafe probability of $p_u = 0.00915 \pm 0.005$ was found. Since the problem can be estimated as a uniform completion problem, the failure probability of the goal network with respect to the unsafe condition chosen can be calculated as a function of the number of time steps the goal network executes:

$$W_s = p_u \times \sum_{x=0}^{c_k} p_n^x, \tag{6.2}$$

where $p_n = 0.99085 \pm 0.005$ is the stationary nominal probability. Given the estimator uncertainties, these failure probabilities are fairly low, which suggests a well-designed system. The failure probability as a function of completion time is shown in Figure 6.21; the failure probability approaches one as the completion time goes to infinity.

6.4 Conclusion

The example goal networks presented in this chapter were very useful in driving the design and improvement of the verification methods introduced in the previous chapters. The rover example in Section 6.2 validated the conversion software and pushed the capabilities of the failure probability calculation procedure. The Titan aerobot example goal network was significantly larger and more

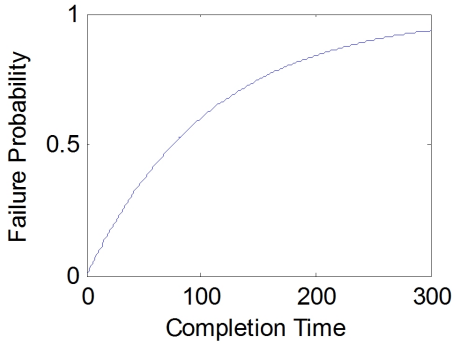


Figure 6.21: Failure probability of the Titan example as a function of completion time

complex and it drove the creation of the SBT Checker and InVeriant verification method due to the inability of the conversion procedure and PHAVer to verify the goal network. Both examples illustrate the techniques in this dissertation well.

Chapter 7

Conclusions and Future Directions

7.1 Summary of Contributions

As autonomous robotic systems are designed to take on more difficult tasks, the complex fault tolerant control systems need to be verified for safety. Model checking is a popular verification approach; several automatic symbolic model checkers are available. However, model checkers for hybrid automata with a continuous state space suffer from the inability to handle the state space explosion that ensues. One way to decrease the effect of the state space explosion is to impose some structure on the robotic control system that allows it to be verified more efficiently.

A goal-based fault tolerant control architecture based on MDS was analyzed in this work. Three methods of safety verification of the goal network control systems, two deterministic and one stochastic, were introduced. The first method was an automatic goal network conversion procedure that connected goal network executions to a hybrid automaton structure via a bisimulation when simple ordering on the goal network's time points was imposed. Then, existing symbolic model checkers, particularly PHAVer, were used to complete the verification. Since the hybrid automaton captured every possible executable set of goals in the goal network as a location, the number of locations was roughly exponential with the number of root goals in a goal network. The conversion procedure, written in Mathematica, was able to convert large goal networks given enough time, however, PHAVer was not always able to verify the converted automaton without state space reductions and abstractions. In general, the number of passive state variables that control failure transitions in the goal network is the limiting factor in PHAVer verification as the state space explosion depends strongly on that. The efficiency of the conversion algorithm also depends strongly on the number of passive state variables because the number of failure transitions from a location grows exponentially with the number of passive state variables constrained in that location.

These complexity issues drove the creation of new design for verification and verification algorithms. When the goal network has state-based transitions, which occurs when each possible passive state satisfies the passive constraints in some set of goal tactics for each group of goals, the invariants of the locations of the converted hybrid automaton contain all the information needed to find every possible transition between the locations. Therefore, the transitions do not need to be found explicitly and the reachability of the locations in a group depends only on the reachability of the states of the passive state variables constrained. This allows the locations only of the converted hybrid automaton to be searched for ones whose invariants and rate conditions satisfy the unsafe conditions. If the passive states constrained in these unsafe locations are reachable from the initial conditions on those state variables, then the unsafe locations are reachable. If one or more of the passive state variables constrained are continuous, rate-driven dependent state variables, such as power or temperature, a path through the discrete sets of states of those state variables from the initial condition to the unsafe condition must be found to prove reachability. However, this path search is simplified by the reachability properties of the automaton due to the state-based transitions requirement. Two software algorithms were introduced, the design for verification tool, SBT Checker, and the verification software, InVeriant. Though these software algorithms were developed for the verification of goal networks, they also can efficiently verify a class of hybrid systems.

The SBT Checker and InVeriant software combination proved to be a more efficient and effective verification method for goal networks. First, the provable modularity of the state-based transitions property allows for distributed design of goal networks. Designers can use the SBT Checker tool to create goal trees that have state-based transitions. The design can be iterative as the software provides nearly instantaneous feedback about which state constraints are missing from the goal tree. The goal network that is the combination of these goal trees is guaranteed to have state-based transitions as long as the controlled constraints are consistent. The state-based transitions and consistent controlled constraints requirements are very useful checks because, instead of being restrictive, they are good design practices that ensure that the tactics in the control system cover every possible modeled passive state. The consistent controlled constraints requirement is checked in the InVeriant algorithm when the locations are created during the goal network conversion. Since the transitions are not created and since most passive state models are not incorporated into the automaton being verified, the complexity issues that result from the number of passive state variables do not affect this algorithm. This allows for larger systems to be verified quickly and effectively. The output of the InVeriant software gives not only the unsafe locations but the set(s) of goals that are

common to them, which aids in the redesign of faulty control systems. For certain unsafe locations, InVeriant will also find an appropriate path to prove reachability.

Fault tolerant systems designed with the concept of state-based transitions are very dependent on the quality of the state estimators for the passive state variables. Methods to calculate the failure probability of a system due to its estimation uncertainty are discussed. These methods are very useful to the design of a system as the result is a measure of how much the tactics depend on faulty estimators or hard to measure state variables, but they are severely limited by complexity issues. However, there are many ways to abstract the problem and there is even an automatic computational algorithm. In many cases, this sort of analysis is ignored, even though it is so important for autonomous systems whose reliance on estimated state values based on sensor measurements is an absolute.

Finally, two significant example goal networks were presented. The complex rover and Titan aerobot examples were verified versus unsafe sets using the conversion/PHAVer method and the SBT Checker/InVeriant method, respectively. These examples tested and improved the conversion procedure and the Titan aerobot example drove the design of the novel verification procedure due to the inability of PHAVer to verify the problem using simple abstractions and model reduction techniques.

This dissertation presents a significant study of the verification of goal-based control programs. The conversion of goal networks to hybrid systems allows for model checking techniques to be applied. A design for verification tool was developed and has great potential for the design and verification of real-world goal-based control systems and linear hybrid automata. The ensuing verification method is efficient enough to handle large goal networks and hybrid systems. Finally, the estimation uncertainty analysis is a novel concept that may lead to useful estimator or goal network control system design techniques.

7.2 Future Directions

There are several directions in which this work can continue. First, the restrictions on time points imposed nearly immediately on the goal network's structure can cause interesting problems in cases where going back or redoing a part of a tactic is required. There may be ways to loosen the time point restrictions imposed while maintaining the important group structure of the goal networks. An important benefit of MDS is its use of projections, which is not included in the goal networks used

for verification. It may be possible to place projection constraints with the rate conditions in each location upon conversion and it may be possible to reason about those constraints in the search for unsafe locations.

The concept of state-based transitions applies to goals networks as well as hybrid automata. When the dynamics in a set of hybrid automata are sufficiently simple, it is possible to use the SBT Checker and InVeriant verification method with the hybrid system directly. In fact, the class of hybrid systems upon which this method could be applied is broader than the class that is bisimilar to goal network control systems. This extra capability of the verification method seems very useful and should be explored more fully.

The two deterministic verification methods include software that is written in Mathematica. Though its kernel is fairly fast, it may be more efficient to convert the software into Java. While the conversion procedure has been tested fairly extensively, the SBT Checker and InVeriant could benefit greatly from more testing, especially on more complex problems. A test of the entire process from design to verification of a control system for a real-world system would be extremely useful.

The failure probability due to state estimation uncertainty is a very important tool in the analysis of a system, and it seems that one should be able to discover very specific feedback on how to redesign a system based on this analysis. Some possible ways to redesign a system include installing better sensors, designing more accurate estimators, or reducing the control dependence on particular state variables. Currently, there is no process or set of guidelines available to aid the analyst in making these design determinations, though this seems to be possible.

The verification emphasis of this work is on safety. Liveness properties, however, are also very important to check. Spin is a model checker that can verify liveness properties of discrete automata. A goal network with unimportant or no continuous state variables can be converted into an automaton specified in Promela code. It can then be verified versus an unsafe Buchi automaton using Spin. However, since Spin is not a symbolic model checker, the state space complexity issues can be even more restrictive. More work could be done to discover a better abstraction of goal networks so that Spin is more effective in their verification. Another benefit of Spin is its non-deterministic execution model, which could make it a good fit for verifying multiple robot systems.

The overall goal of this work and others on verification of autonomous control systems is to find an effective way to design these systems so that they work in all foreseeable situations (and some that are not). It is the author's belief that verification work must begin during the requirements stage of the system design and continue throughout the creation of the autonomous system. Therefore, the

tools and concepts of model checking must be integrated into the system architecture and design; at the same time, there must be enough flexibility to allow for complex behaviors to emerge in the autonomous systems. While the goal structure imposed in this work does not have the necessary flexibility, some important concepts, particularly the modularity of state-based control transitions, have been discovered and will be influential in future work towards the design of robust, fault-tolerant autonomous control systems.

Appendix A

DTD Files

The Document Type Definition (DTD) files for the XML input files for PHAVer and Spin are presented here. These DTD files list out the necessary and possible elements for the given type of XML file. These files create a standard for the XML input files; they check that the necessary information is entered in the proper way.

A.1 PHAVer

```
<?xml version="1.0"?>

<!ELEMENT inputs (smc,cdsv,usv,goals,tcs,unsafe)>
<!ELEMENT smc (#PCDATA)>

<!ELEMENT cdsv (sv+)>
<!ELEMENT sv (name, constraint+, initial_condition)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT constraint (type_name,merge+,trans,dyn_eqn,reset?)>
<!ELEMENT type_name (#PCDATA)>
<!ELEMENT merge (constraint_type,merge_condition,merge_type?,
                  merged_constraint*)>
<!ELEMENT constraint_type (#PCDATA)>
<!ELEMENT merge_condition (#PCDATA)>
<!ELEMENT merge_type (#PCDATA)>
<!ELEMENT merged_constraint (#PCDATA)>
```

```
<!ELEMENT trans (entrylogic,exitlogic)>
<!ELEMENT entrylogic (#PCDATA)>
<!ELEMENT exitlogic (#PCDATA)>
<!ELEMENT dyn_eqn (#PCDATA)>
<!ELEMENT reset (#PCDATA)>
<!ELEMENT initial_condition (#PCDATA)>

<!ELEMENT usv (svu+)>
<!ELEMENT svu (name,trans_type,parameter*,input_var*,transition*,
               dynamics+,initial_condition)>
<!ELEMENT trans_type (#PCDATA)>
<!ELEMENT input_var (#PCDATA)>
<!ELEMENT parameter (#PCDATA)>
<!ELEMENT transition (start,end,condition)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT end (#PCDATA)>
<!ELEMENT condition (#PCDATA)>
<!ELEMENT dynamics (value,eqn,reset?,condition)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT eqn (#PCDATA)>


<!ELEMENT goals (goal+,pgoal+)>
<!ELEMENT goal (name,tp_start,tp_end,constraint_sv?,constraint_type?,
               constraint_value*,tactic*)>
<!ELEMENT pgoal (name,tp_start,tp_end,constraint_sv,constraint_type,
                   constraint_value)>
<!ELEMENT tp_start (#PCDATA)>
<!ELEMENT tp_end (#PCDATA)>
<!ELEMENT constraint_sv (#PCDATA)>
<!ELEMENT constraint_value (#PCDATA)>
<!ELEMENT tactic (goalname+,failure*)>
```

```
<!ELEMENT goalname (#PCDATA)>
<!ELEMENT failure (logic,destination)>
<!ELEMENT logic (#PCDATA)>
<!ELEMENT destination (#PCDATA)>

<!ELEMENT tcs (tconstraint*)>
<!ELEMENT tconstraint (start,end,min,max)>
<!ELEMENT max (#PCDATA)>
<!ELEMENT min (#PCDATA)>

<!ELEMENT unsafe (uset+)>
<!ELEMENT uset (ucons+)>
<!ELEMENT ucons (svc,type,constr)>
<!ELEMENT svc (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT constr (#PCDATA)>}
```

A.2 Spin

```
<?xml version="1.0"?>

<!ELEMENT inputs (smc,cdsv,usv,goals,tcs)>
<!ELEMENT smc (#PCDATA)>

<!ELEMENT cdsv (dt?,sv+)>
<!ELEMENT dt (#PCDATA)>
<!ELEMENT sv (name, constraint+, initial_condition)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT constraint (type_name,merge+,trans,dyn_eqn,reset?)>
<!ELEMENT type_name (#PCDATA)>
<!ELEMENT merge (constraint_type,merge_condition,merge_type?,
                    merged_constraint*)>
```

```
<!ELEMENT constraint_type (#PCDATA)>
<!ELEMENT merge_condition (#PCDATA)>
<!ELEMENT merge_type (#PCDATA)>
<!ELEMENT merged_constraint (#PCDATA)>
<!ELEMENT trans (entrylogic,exitlogic)>
<!ELEMENT entrylogic (#PCDATA)>
<!ELEMENT exitlogic (#PCDATA)>
<!ELEMENT dyn_eqn (#PCDATA)>
<!ELEMENT reset (#PCDATA)>
<!ELEMENT initial_condition (#PCDATA)>

<!ELEMENT usv (svu+)>
<!ELEMENT svu (name,trans_type,input_var*,transition*,dynamics+,
               initial_condition)>
<!ELEMENT trans_type (#PCDATA)>
<!ELEMENT input_var (#PCDATA)>
<!ELEMENT transition (start,end,condition)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT end (#PCDATA)>
<!ELEMENT condition (#PCDATA)>
<!ELEMENT dynamics (value,eqn?,reset?,condition?)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT eqn (#PCDATA)>


<!ELEMENT goals (goal+)>
<!ELEMENT goal (name,tp_start,tp_end,constraint_sv?,constraint_type?,
               constraint_value*,tactic*)>
<!ELEMENT tp_start (#PCDATA)>
<!ELEMENT tp_end (#PCDATA)>
<!ELEMENT constraint_sv (#PCDATA)>
<!ELEMENT constraint_value (#PCDATA)>
```

```
<!ELEMENT tactic (goalname+,startsin,failure+)>
<!ELEMENT goalname (#PCDATA)>
<!ELEMENT startsin (#PCDATA)>
<!ELEMENT failure (logic,destination)>
<!ELEMENT logic (#PCDATA)>
<!ELEMENT destination (#PCDATA)>

<!ELEMENT tcs (tconstraint*)>
<!ELEMENT tconstraint (start,end,min,max)>
<!ELEMENT max (#PCDATA)>
<!ELEMENT min (#PCDATA)>
```

Glossary

branch goal goal with no child goals in group. 35

compatible goals that are not elaborated into different tactics of the same parent goal. 31

complete system state state that includes the actual and estimated state of each uncertain state variable. 77

completion goal controlled goal with a transition constraint. 11

completion time minimum length of a nominal execution path for a group. 81

consistent constraints that can be executed concurrently. 31

contribution value normalized contribution of a location towards a completion task. 82

controllable state variable state variable associated directly with a command class. 21

controlled goal constraint that causes a command to be issued to the system. 10

dependent state variable state variable not associated with a command class but dependent on controllable or dependent state variables. 21

elaboration act of choosing a control tactic out of those available. 9

executable branch set of goals in a goal tree that can be executed concurrently. 61

executable set set of goals that can execute concurrently. 32

failure path execution path that includes a complete system state that is unsafe. 82

goal constraint on a state variable in time. 9

group set of goals that are active between consecutive time points or the corresponding set of locations. 30

linear hybrid automaton systems with discrete modes of execution that have different continuous behavior. 15

location discrete mode of execution. 15

nominal execution path set of nominal complete system states that represents the complete, safe execution of a group. 81

non-uniform completion group whose execution time depends on the states visited. 82

passive goal constraint with no associated command. 10

root goal goal with no parent. 24

sibling goal goals elaborated into the same tactic. 24

state variable states of the system or environment. 9

state-based transitions goal network or hybrid system that has control tactics or modes for every modeled passive state. 58

tactic control mode or method. 11

uncontrollable state variable state variable not associated with any command class and not dependent on any controllable or dependent state variable. 21

uniform completion group whose execution time does not depend on the states visited. 82

unsafe set set of conditions that should never be reached by a goal network or hybrid system execution. 79

Bibliography

[1] A. Elfes, J. L. Hall, J. F. Montgomery, C. F. Bergh, and B. A. Dudik, "Towards a substantionally autonomous aerobot for exploration of Titan," in *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 2535–2541, 2004.

[2] J. W. Burdick, N. E. Du Toit, A. Howard, C. Looman, J. Ma, R. M. Murray, and T. Wongpiromsarn, "Sensing, navigation and reasoning technologies for the DARPA Urban Challenge," tech. rep., DARPA Urban Challenge Final Report, 2007.

[3] S. Croomes, "Overview of the DART Mishap Investigation Results," tech. rep., National Aeronautics and Space Administration, 2006.

[4] L. B. Cremean, T. B. Foote, J. H. Gillula, G. H. Hines, D. Kogan, K. L. Kriechbaum, J. C. Lamb, J. Leibs, L. Lindzey, A. D. Stewart, J. W. Burdick, and R. M. Murray, "Alice: An information-rich autonomous vehicle for high-speed desert navigation," *Journal of Field Robotics*, vol. 23, pp. 777–810, 2006.

[5] P. S. Morgan, "Fault protection techniques in JPL spacecraft," in *Proc. of the First International Forum on Integrated System Health Engineering and Management in Aerospace (ISHEM)*, 2005.

[6] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[7] P. Abbeel, A. Coates, M. Montemerlo, A. Y. Ng, and S. Thrun, "Discriminative training of Kalman filters," in *Proc. of Robotics: Science and Systems*, 2005.

[8] P. Goel, G. Dedeoglu, S. I. Roumeliotis, and G. S. Sukhatme, "Fault detection and identification in a mobile robot using multiple model estimation and neural network," in *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 2302–2309, 2000.

[9] M. W. Hofbaur and B. C. Williams, "Hybrid diagnosis with unknown behavioral modes," in *Proc. of the 13th International Workshop on Principles of Diagnosis*, 2002.

[10] V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Real-time fault diagnosis [robot fault diagnosis]," *IEEE Robotics and Automation Magazine*, vol. 11, no. 2, pp. 56–66, 2004.

[11] K. Ben Lamine and F. Kabanza, "History checking of temporal fuzzy logic formulas for monitoring behavior-based mobile robots," in *Proc. of the 12th IEEE International Conference on Tools with Artificial Intelligence*, 2000.

[12] M. Blanke, M. Staroswiecki, and N. E. Wu, "Concepts and methods in fault-tolerant control," in *Proc. of the American Control Conference*, 2001.

[13] C. Ferrell, "Failure recognition and fault tolerance of an autonomous robot," *Adaptive Behaviour*, vol. 2, no. 4, pp. 375–398, 1994.

[14] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, "A dynamic fault tolerance framework for remote robots," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 4, pp. 477–490, 1995.

[15] T. C. Lueth and T. Laengle, "Fault-tolerance and error recovery in an autonomous robot with distributed controlled components," in *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 8–13, Springer-Verlag, 1994.

[16] L. E. Parker, "ALLIANCE: An architecture for fault tolerant multirobot cooperation," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 2, pp. 220–240, 1998.

[17] Y. Diao and K. M. Passino, "Intelligent fault-tolerant control using adaptive and learning methods," *Control Engineering Practice*, vol. 10, pp. 801–817, 2002.

[18] Y. Zhang and J. Jiang, "Fault tolerant control system design with explicit consideration of performance degradation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, pp. 838–848, July 2003.

[19] P. Kim, B. C. Williams, and M. Abramson, "Executing reactive model-based programs through graph-based temporal planning," in *Proc. of the International Joint Conference on Artificial Intelligence*, 2001.

[20] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, "Model-based programming of fault-aware systems," *AI Magazine*, vol. 24, no. 4, pp. 61–75, 2003.

[21] B. C. Williams, P. Kim, M. Hofbaur, J. How, J. Kennell, J. Loy, R. Ragno, J. Stedl, and A. Walcott, "Model-based reactive programming of cooperative vehicles for Mars exploration," in *Proc. of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001.

[22] M. D. Ingham and B. C. Williams, "Timed model-based programming: Executable specifications for robust critical sequences," in *Proc. of the International Workshop on Self-Adaptive Software*, 2003.

[23] M. J. Mataric, "Integration of representation into goal-driven behavior-based robots," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 3, pp. 304–312, 1992.

[24] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, 1986.

[25] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software architecture themes in JPLs Mission Data System," in *Proc. of the IEEE Aerospace Conference*, 2000.

[26] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada, "Engineering complex embedded systems with State Analysis and the Mission Data System," *AIAA Journal of Aerospace Computing, Information and Communication*, vol. 2, pp. 507–536, December 2005.

[27] R. D. Rasmussen, "Goal-based fault tolerance for space systems using the Mission Data System," in *Proc. of the IEEE Aerospace Conference*, vol. 5, pp. 2401–2410, March 2001.

[28] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[29] E. Klavins, "A formal model of a multi-robot control and communication task," in *Proc. of the 42th IEEE Conference on Decision and Control*, 2003.

[30] A. E. Haxthausen and J. Peleska, "Formal development and verification of a distributed railway control system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 687–701, 2000.

[31] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–126, 1995.

[32] T. Ball and S. K. Rajamani, *SPIN*, vol. LNCS 1885, ch. Bebop: A Symbolic Model Checker for Boolean Programs, pp. 113–130. Springer-Verlag, 2000.

[33] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *CAV*, vol. LNCS 2404, ch. NuSMV 2: An Open Source Tool for Symbolic Model Checking, pp. 359–364. Springer-Verlag, 2002.

[34] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 428–439, 1990.

[35] F. Schneider, S. Easterbrook, J. Callahan, and G. Holzmann, "Validating requirements for fault tolerant systems using model checking," in *Proc. of the Third International Conference on Requirements Engineering*, pp. 4–13, 1998.

[36] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *TACAS/ETAPS*, vol. LNCS 1579, ch. Symbolic Model Checking without BDDs, pp. 193–207. Springer-Verlag, 1999.

[37] K. L. McMillan, *CAV*, vol. LNCS 2404, ch. Applying SAT Methods in Unbounded Symbolic Model Checking, pp. 250–264. Springer-Verlag, 2002.

[38] R. Alur, T. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 181–201, 1996.

[39] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, 1997.

[40] K. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[41] D. Dill and H. Wong-Toi, *CAV 95: Computer-aided Verification*, ch. Verification of real-time systems by successive over and under approximation, pp. 409–422. Springer, 1995.

[42] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Proc. of the International Conference on Hybrid Systems: Computation and Control*, 2005.

[43] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[44] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, no. 1, pp. 110–121, 2005.

[45] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "State-space reduction techniques in agent verification," in *Proc. of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 896–903, 2004.

[46] E. Haghverdi, P. Tabuada, and G. J. Pappas, "Bisimulation relations for dynamical, control, and hybrid systems," *Theoretical Computer Science*, vol. 342, no. 2-3, pp. 229 – 261, 2005.

[47] P. Tabuada and G. J. Pappas, "Bisimilar control affine systems," *Systems and Control Letters*, vol. 52, no. 1, pp. 49 – 58, 2004.

[48] A. Girard and G. J. Pappas, "Approximate bisimulation relations for constrained linear systems," *Automatica*, vol. 43, no. 8, pp. 1307 – 1317, 2007.

[49] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, *Programming Multi-Agent Systems*, vol. LNAI 3067, ch. Verifiable Multi-agent Programs, pp. 72–89. 2004.

[50] T. Suzuki, S. M. Shatz, and T. Murata, "A protocol modeling and verification approach based on a specification language and petri nets," *IEEE Transactions on Software Engineering*, vol. 16, no. 5, pp. 523–536, 1990.

[51] R. Simmons, C. Pecheur, and G. Srinivasan, "Towards automatic verification of autonomous systems," in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, pp. 1410–1415, 2000.

[52] A. Holt, "Formal verification with natural language specifications: guidelines, experiments and lessons so far," *South African Computer Journal*, no. 24, pp. 253–257, 1999.

[53] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, "Specification styles in distributed systems design and verification," *Theoretical Computer Science*, vol. 89, pp. 179–206, 1991.

[54] R. S. Beata Sarna-Starosta and L. K. Dillon, "A model-based design-for-verification approach to checking for deadlock in multi-threaded applications," in *Proc. of 18th International Conference on Software Engineering and Knowledge Engineering*, 2006.

[55] A. Betin-Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp, "Application of design for verification with concurrency controllers to air traffic control software," in *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 14–23, ACM, 2005.

[56] N. Sharygina, J. C. Browne, and R. P. Kurshan, "A formal object-oriented analysis for software reliability: Design for verification," in *Proc. of the Fundamental Approaches to Software Engineering Conference*, pp. 318–332, 2001.

[57] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh, "Assume-guarantee verification of source code with design-level assumptions," in *Proc. of the 26th International Conference on Software Engineering*, pp. 211–220, IEEE Computer Society, 2004.

[58] T. Bultan and A. Betin-Can, *Verified Software: Theories, Tools, Experiments*, vol. LNCS 4171, ch. Scalable Software Model Checking Using Design for Verification, pp. 337–346. Springer-Verlag, 2008.

[59] G. De Giacomo and M. Y. Vardi, *ECP-99*, vol. LNAI 1809, ch. Automata-Theoretic Approach to Planning for Temporally Extended Goals, pp. 226–238. 2000.

[60] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Wheres Waldo? Sensor-based temporal logic motion planning," in *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 3116–3121, 2007.

[61] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[62] V. Gupta, R. Jagadeesan, and P. Panangaden, "Stochastic processes as concurrent constraint programs," in *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 189–202, ACM, 1999.

[63] H. L. S. Younes, *CAV*, vol. LNCS 3576, ch. Probabilistic Verification for "Black-Box" Systems, pp. 253–265. Springer-Verlag, 2005.

[64] M. Kwiatkowska, "Model checking for probability and time: from theory to practice," in *Proc. of the 18th Annual IEEE Symposium on Logic in Computer Science*, pp. 351–360, June 2003.

[65] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time markov chains," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 162–170, 2000.

[66] J. Sproston, "Decidable model checking of probabilistic hybrid automata," in *Proc. of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 31–45, Springer-Verlag, 2000.

[67] S. Prajna, A. Jadbabaie, and G. J. Pappas, "Stochastic safety verification using barrier certificates," in *Proc. of the IEEE Conference on Decision and Control*, 2004.

[68] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: a hybrid approach," *International Journal on Software Tools Technology Transfer*, vol. 6, pp. 128–142, 2004.

[69] X. Koutsoukos and D. Riley, "Computational methods for verification of stochastic hybrid systems," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 38, pp. 385–396, March 2008.

[70] S. Amin, A. Abate, M. Prandini, J. Lygeros, and S. Sastry, "Reachability analysis for controlled discrete time stochastic systems," in *Proc. of the International Conference on Hybrid Systems: Computation and Control*, pp. 49–63, 2006.

[71] A. Abate, M. Prandini, J. Lygeros, and S. Sastry, "Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems," *Automatica*, vol. 44, no. 11, pp. 2724 – 2734, 2008.

[72] H. A. P. Blom, G. Bakker, and J. Krystul, "Probabilistic reachability analysis for large scale stochastic hybrid systems," in *Proc. of the 46th IEEE Conference on Decision and Control*, pp. 3182–3189, 2007.

[73] S. K. Au and J. L. Beck, "Estimation of small failure probabilities in high dimensions by subset simulation," *Journal of Probabilistic Engineering Mechanics*, vol. 16, 2001.

[74] S. P. Brooks, "Markov Chain Monte Carlo method and its applications," *The Statistician*, vol. 47, no. 1, pp. 69–100, 1998.

[75] C. J. Geyer, "Practical Markov Chain Monte Carlo," *Statistical Science*, vol. 7, no. 4, pp. 473–483, 1992.

[76] D. Dvorak, R. Rasmussen, and T. Starbird, "State knowledge representation in the Mission Data System," in *Proc. of the IEEE Aerospace Conference*, 2002.

[77] D. Dvorak, M. Indictor, M. Ingham, R. Rasmussen, and M. Stringfellow, "A unifying framework for systems modeling, control systems design, and system operation," in *Proc. of the IEEE Conference on Systems, Man, and Cybernetics*, October 2005.

[78] D. Dvorak, "Challenging encapsulation in the design of high-risk control systems," in *Proc. of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2002.

[79] G. Labinaz, M. M. Bayoumi, and K. Rudie, "A survey of modeling and control of hybrid systems," *Annual Reviews of Control*, vol. 21, pp. 79–92, 1997.

[80] G. Pola, M. L. Bujorianu, J. Lygeros, and M. D. D. Benedetto, "Stochastic hybrid systems: An overview," in *Proc. of the IFAC Conference on Analysis and Design of Hybrid Systems*, 2003.

[81] J. Hu, J. Lygeros, and S. Sastry, "Towards a theory of stochastic hybrid systems," in *Proc. of the Hybrid Systems: Computation and Control* (N. Lynch and B. Krogh, eds.), vol. Lecture Notes in Computer Science 1809, pp. 160–173, Springer, 2000.

[82] A. Elfes, J. F. Montgomery, J. L. Hall, S. S. Joshi, J. Payne, and C. F. Bergh, "Autonomous flight control for a Titan exploration aerobot," in *Proc of Robotics Science and Systems*, 2005.