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

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

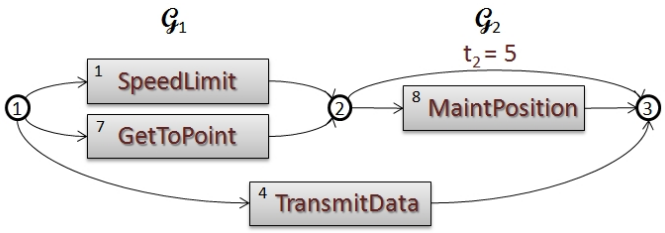

Figure 3.1: Goal network with two groups

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

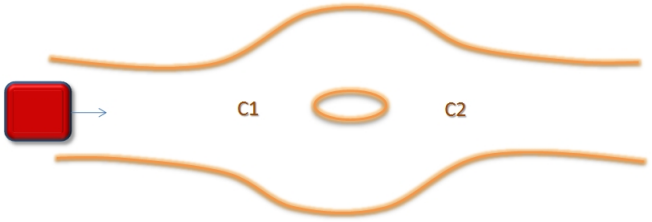

Figure 3.2: Path for the simple rover example

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.

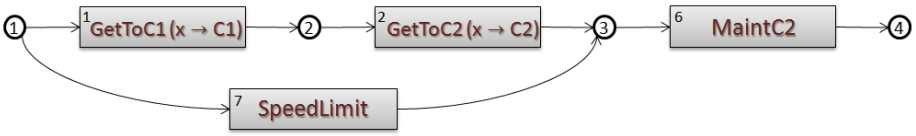

Figure 3.3: Goal network for the simple rover example

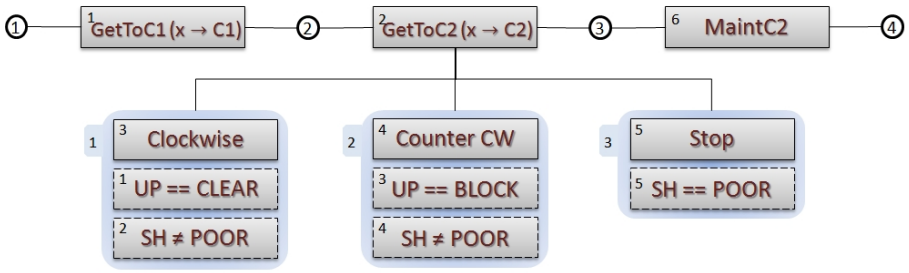

Figure 3.4: Route goal trees for the simple rover example

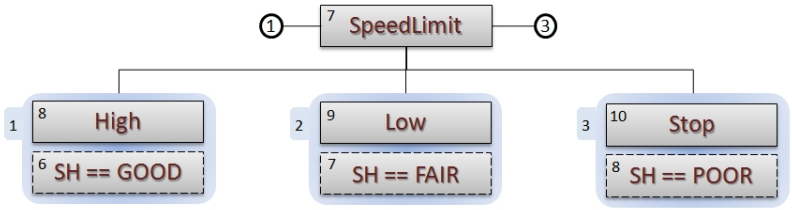

Figure 3.5: Speed limit goal tree for the simple rover example

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.

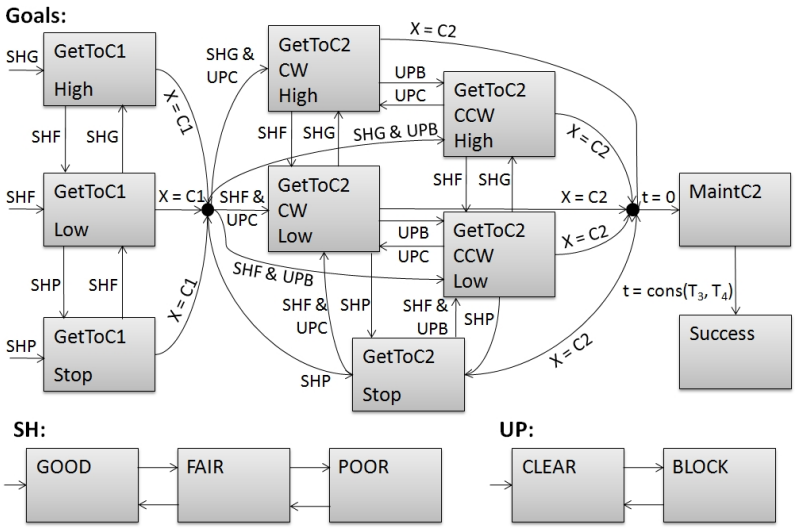

Figure 3.6: Automata for rover example

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-

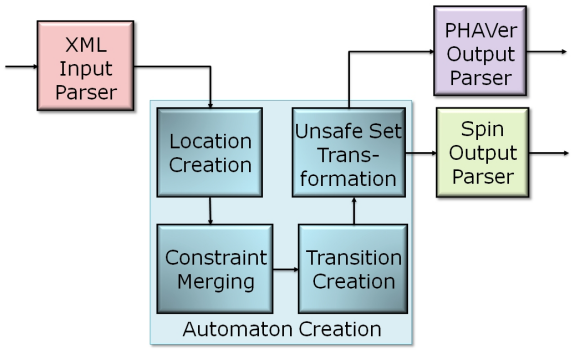

Figure 3.7: Flow chart of the conversion software execution

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.

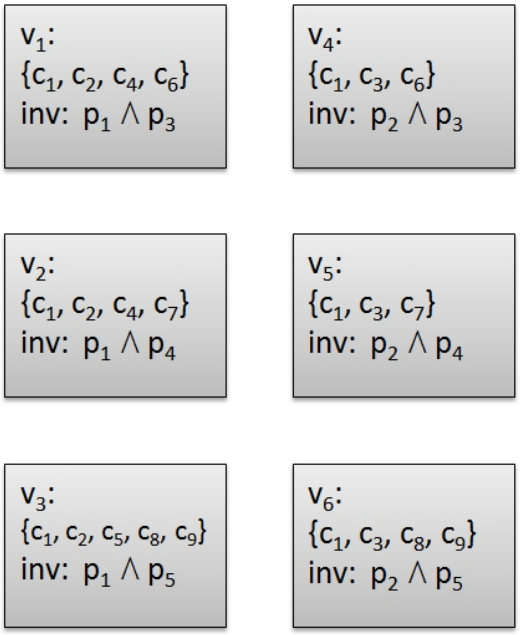

Figure 3.8: Simple hybrid system for reverse conversion example. Transitions are omitted for clarity.

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	$\{v_3, v_6\}$		

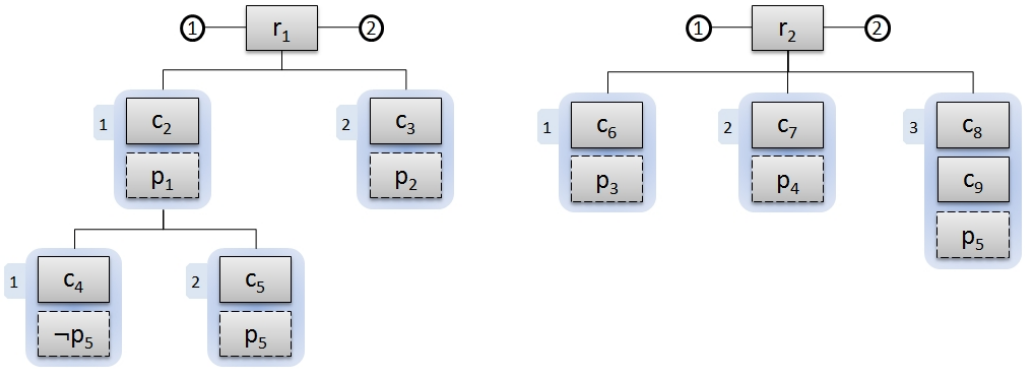

Figure 3.9: Converted goal trees for reverse conversion example

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.