Automatic Conversion Software for the Safety Verification of Goal-based Control Programs

Julia M. B. Braman and Richard M. Murray

***Abstract*— Fault tolerance and safety verification of control systems are essential for the success of autonomous robotic systems. A control architecture called Mission Data System (MDS), developed at the Jet Propulsion Laboratory, takes a goal-based control approach. In this paper, an automatic software algorithm for converting goal network control programs into linear hybrid systems is described. The linear hybrid system can then be verified for safety in the presence of failures using existing symbolic model checkers. Several benchmark problems are converted from goal networks to linear hybrid automata, illustrating that complex goal networks can be converted and verified using this method.**

I. Introduction

Autonomous robotic missions by nature have complex control systems. In general, the necessary fault detection, isolation and recovery software for these 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. One way to accomplish this could be to create a flexible control system that can reconfigure itself in the presence of faults. However, if the control system cannot be verified for safety, the added complexity of reconfigurability could reduce the system's effective fault tolerance.

One particularly useful way to model fault tolerant control systems is as hybrid systems. Much work has been done on the control of hybrid systems [1]. 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 [2]. There are several software packages available that can be used for this analysis, including HyTech [3], UPPAAL [4], and VERITI [5], all of which are symbolic model checkers. PHAVer, the symbolic model checker used in this paper, is a more capable extension of HyTech [6]. PHAVer 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. Safety verification for fault tolerant hybrid control systems ensures that the occurrence of certain faults will not cause the system to reach an unsafe state.

Often times, the control program used for an application is not in a form that is conducive to verification. The control program must then be transformed to an acceptable form by some suitable means. 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 [7], and Java, for which the JPF2 is a general purpose model checker [8]. Another popular approach is to create correct by design control programs from Buchi automata on infinite words [9], or from specifications and requirements stated in a restricted subset of LTL [10].

The software control architecture that the control programs in this paper are based on is Mission Data System (MDS), developed at the Jet Propulsion Laboratory [11]. MDS is based on a systems engineering concept called State Analysis [12]. Systems that use MDS 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 [13].

In this paper, an automated process to convert goal networks to linear hybrid systems is presented. A procedure that allows a user to work through the conversion by hand is found in a previous work [14]. This procedure and some supporting information about State Analysis and linear hybrid systems is briefly described in Section II. The automated software version of the conversion process that allows more complicated goal networks to be correctly converted into linear hybrid systems is discussed in Section III. Benchmark examples are introduced in Section IV, and conclusions and future work are presented in Section V.

II. Background Information

A. 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 [12]. Models of state effects in the system that is under control are used for such things as 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. 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.

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 [12]. 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

J. Braman and R. Murray are with the Dept. of Mech. Eng., California Institute of Technology, Pasadena, CA 91125, USA braman@caltech.edu

control system. Therefore, the software architecture, MDS, is closely related to State Analysis.

Data structures called software state variables are central to MDS [15]. A state variable can contain much information; for example, a position state variable for a robot in the plane could contain the robot's (x, y) position, its velocity in component form, and uncertainty values for each piece of information. Each state variable has a unique estimator, and if necessary, a controller. Goals can be created that constrain some or all of a state variable's information. For example, a goal could constrain the velocity of the position state variable used in the previous example, but could leave the position or uncertainties unconstrained.

Goal networks replace command sequences as the control input to the system. A goal network consists of a set of goals with their associated starting and ending time points and temporal constraints. Temporal constraints on a goal may be based on execution time, or based on the completion of the goal. 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.

Elaboration allows MDS to handle tasks more flexibly 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 class for a goal can include several pre-defined tactics. These tactics are simply different ways to accomplish the intent of the goal, and tactics may be logically chosen by the elaborator based on programmer-defined conditions. This capability allows for many common types and combinations of faults to be accommodated automatically by the control system [13].

B. Linear Hybrid Automata

There are several symbolic model checkers available that are capable of verifying linear hybrid automata. A linear hybrid automaton H consists of the following components [3]:

1) A finite, ordered list of continuous state variables, $X = \{x_1, x_2, ..., x_n\}$, and their time derivatives, $\dot{X} = \{\dot{x_1}, \dot{x_2}, ..., \dot{x_n}\}$.
2) 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.
3) The set of invariants for each location, *inv*(v), the set of initial conditions for each location, *init*(v), and the set of flow conditions for each location, *dif*$(v, \dot{X})$, where $v \in V$.
4) The set of transition labels, Σ, and transition actions or reset equations, A.

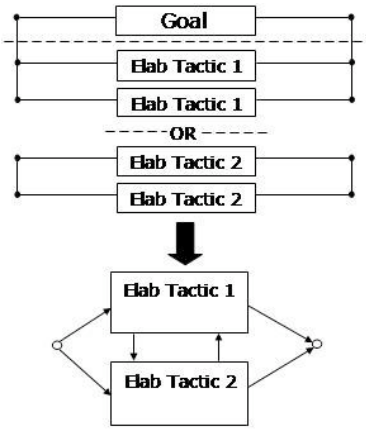

Fig. 1. Representation of a goal network to hybrid system conversion. The goals are represented by rectangles between circular time points, and elaboration is depicted by a dashed line, with child goals below the line. In the hybrid automaton, the circles represent group connectors and rectangles represent locations.

5) The set S of synchronization labels.

The above components fully 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. PHAVer deals with this explosion by partitioning locations into simpler ones, or by using overapproximation of sets of locations to limit the complexity of the system and to accelerate convergence and force termination by approximating a system where reachability is decidable [6]. In analyses where overapproximation is not required, an exact safety guarantee is possible.

C. Goal Network Conversion and Verification Procedure

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. A manual process for converting certain types of goal networks is described in [14], and is graphically represented in Figure 1. 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.

Each state variable in the goal network is labeled as either controllable, uncontrollable, or dependent. A controllable state variable (CSV) is directly associated with a command class. An uncontrollable state variable (USV) is not associated with a command class in any way. A dependent state variable (DSV) has model dependencies on at least one controllable state variable. Different hybrid automata are created from goals on and states of these different types of state variables.

An outline of the conversion procedure for the goals on CSVs and DSVs is as follows:

1) Create elaboration and transition logic tables for each goal that elaborates any constraints on CSVs and for each CSV and continuous DSV, respectively. List transition conditions between states for each discrete DSV.
2) Place goals between consecutive time points into groups.
3) In each group, create locations (modes) by combining branch goals (goals on CSVs that are not ancestors of other goals on CSVs in the group) with all parent and sibling goals (goals in the same tactic or other root goals) that constrain CSVs. Label each location with the dynamical update equations for all CSVs and continuous DSVs constrained in the location. Create Success and Safing locations.
4) Create transitions between locations and groups using the elaboration and transition logic tables found above. Elaboration logic controls transitions into groups and failure transitions between locations in a group, and transition logic controls the transitions out of a group to the next group or to the Success location.
5) Add exit and failure transitions based on time to locations containing goals that have time constraints. Add entry actions that reset the time variable to zero when transitions into these locations from the group connector are taken.
6) Remove unnecessary locations, groups, and transitions.

For each each USV, a separate hybrid automaton is created by making locations from the discrete states or discrete sets of states of the variable. The transition conditions are stochastic rates or are based on the state models. For safety verification, the hybrid automata are converted into PHAVer code and the appropriate transitions are synchronized between the automata. The unsafe (or incorrect) set is determined and conditions that would cause the hybrid automata to enter the unsafe set are searched for using the verification software. If no such conditions are found, the goal network is said to be verified.

III. Conversion Software Design

The software version of the goal network conversion procedure is not yet as comprehensive as the version outlined in the previous section, however most of the important capabilities are present. The conversion software is written in Mathematica because of the list structure it employs and its extensive library of pattern-matching functions.

A. Software Inputs

The inputs to the conversion software are derived from the necessary design components needed to have an implementable goal network in MDS. A set of five lists are needed, and they are $\{goals, usv, merge, elab, trans\}$.

The $goals$ list is an ordered list of all the goals on controllable and dependent state variables in the goal network. The goals must be ordered by ancestory; no goal can appear in the list ahead of its parent. As goal elaboration is a top-down procedure, this constraint on the goal ordering is trivial to achieve. Each $goals$ entry must have the following information listed in this order:

1) Goal number - this is assigned according to the ancestory constraint only.
2) Time point list, $\{t_s, t_e\}$ - this sublist has two elements, the number of the beginning time point and the number of the ending time point. The time points are numbered by order of execution and it is always true that $t_s < t_e$.
3) Parent goal number - this is zero if the goal has no ancestor.
4) Tactic number - this is zero if the goal has no parent.
5) Constraint list, $\{sv, type, \{c_1, ..., c_n\}\}$ - the first element of this sublist is the numerical representation of the controllable or dependent state variable (the set of controllable state variables is ordered arbitrarily, followed by an arbitrary ordering of dependent state variables), the number of the constraint type for that state variable (also ordered arbitrarily), and the set of n constraints that accompany that constraint type.

The usv list is an ordered list of all of the uncontrollable state variables that are constrained in the goal network. The ordering of the uncontrollable state variables is arbitrary. Each usv entry must have the following information listed in this order:

1) State variable name - this is a symbol that will be used in all elaboration and failure logic. The symbol must be unique for each uncontrollable state variable.
2) Discrete value list, $\{v_1, ..., v_m\}$ - this sublist contains the m discrete values or sets of values that the uncontrollable state variable can take. The m value names can be numerical or strings, and must be unique for that specific USV, but not across all USVs.
3) Transition list, $\{\{\{c, v_i\}, ...\}, ..., \{\{c, v_j\}, ...\}\}$ - this list has m sublists, one for each discrete value that the USV can take. In each sublist, there is a set of lists that describe all the transitions that can be taken out of that state value. The first element of those lists, c, is the transition condition, and the second element is the value label to which the transition goes.

The $merge$ list has a sublist for every controllable and dependent state variable in numerical order. These sublists have several parts that describe the type of goal constraints that can be placed on them and how these constraint types merge with one another if two or more constraints on the same state variable are active at the same time. Inside the sublists, there are lists of merge conditions for each constraint type with itself and with each constraint type whose numerical label is larger than its own. The parts of these lists are as follows:

1) Merge type - if the merge is conditional on the constraints, this value is -1. If the merge is always impossible, the value is 0, and there are no more list elements. If the merge is always possible, this value is the constraint type number that the merged constraint becomes.
2) Merge rules for constraints - these are symbolic and are

replaced by actual constraint values in the procedure. This is the last entry for merges that are always possible.

3) Merge conditions - these are also symbolic, and the actual constraint values must be substituted during the merging procedure. If the conditions are true, the merge happens.
4) Constraint type - for conditional merges only, this is the number of the constraint type that the merged constraint becomes.

The $elab$ list has a sublist for every goal listed in the $goals$ list. The sublists each have lists for different logic types and conditions if the goal is a parent goal. If not, the sublist for that goal has only a zero. These elaboration logic information types are listed below:

1) "Starts in" logic - this lists for each tactic the conditions that cause the elaboration of one tactic over another.
2) "Failure" logic - this lists for each tactic the conditions that would fail a tactic and to which goal (or a representative goal, if execution goes to another tactic with more than one goal) the tactic fails. There may be more than one set of conditions and goals for each tactic. The tactic may fail to "Safe" and a failure condition may be the failure of children goals, labeled "CGF."

The $trans$ list has a sublist for every controllable and dependent state variable. The lists in the sublists have transitions conditions must be true for each constraint type for that controllable or dependent state variable. These conditions have two components, as follows:

1) Incoming transition logic - these conditions must be true for this type of constraint to be entered.
2) Outgoing transition logic - these conditions signify the completion of a constraint type.

B. Software Outputs

Since all the necessary information (except flow) for the USV automata are inputs to the software, the output from the software has information only about the CSV/DSV goal automaton. This structure, called $cdsva$, has many parts, organized first into groups and second into locations. In each location's list, the following information is present in order:

1) List of goals present - listed by goal number.
2) List of merged constraints - including CSV and DSV number, type of merged constraint, and constraint values.
3) "Starts in" transition condition - this transition is from the group connector between the previous group and the current group, so only the condition is listed.
4) Incoming and outgoing transition conditions - these transitions are from the preceding group connector to the location, and from the location to the following group connector, respectively. The conditions involve the CSVs and DSVs constrained in that location.

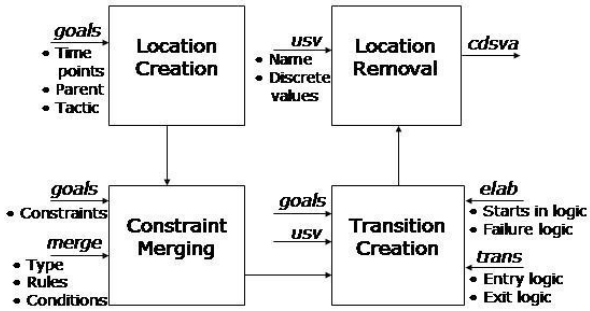

Fig. 2. Flow chart of the conversion software execution

5) Outgoing failure transitions - both the condition and the accepting location are listed. There may be more than one per location.
6) Incoming failure transitions - both the condition and the originating location are listed. There may be more than one per location.

C. Conversion Algorithms

The general outline for the structure of the conversion software is shown in Figure 2. There are four main parts to the conversion algorithm: location creation, constraint merging, transition creation, and location removal. Each of these algorithms is described in this section.

The location creation algorithm takes the $goals$ input structure and uses the time point, parent, and tactic information to sort the goals into groups and then into locations. The time point information is first used to sort the goals into groups, where some goals may occur in more than one group if $t_e > t_s + 1$. Then, the parent and tactic information is used to find all incompatible goals for each goal in the group. These incompatible goals cannot occur in the same location as the goal because they are either in different tactics elaborated from the same parent goal or are descended from goals that are in different tactics from the same parent goal. Next, branch goals for each group are found and each is made into a location; sibling branch goals are combined into one location. Branch goals have no children in the group, and sibling goals are goals that are elaborated into the same tactic or are root goals (goals with no ancestors in the group) that have no common ancestors. Parent goals and their siblings are added to each location until the root goals are reached. Each location is then combined with each other compatible location and repeat locations are removed until no more locations can be combined or removed. Compatible locations are locations that share two or more goals in common and no goals in one location are on the incompatible goal list of any goal in the other location. These steps ensure that each possible execution of the goal network between two time points is represented in exactly one location.

The constraint merging algorithm uses the location output from the location creation algorithm and the $goals$ and $merge$ inputs to combine the constraints on all of the controllable and dependent state variable constrained in each

location. The constraint types of goals on the same state variable are compared to find the proper field of the $merge$ input, and then the symbolic merge conditions and merged constraint values are replaced with the actual constraints found in the $goals$ input. If the merge is successful, the new merge type and merged constraints are added to the location. If the merge is not successful, the location is removed.

The transition creation algorithm creates the transitions from the preceding group connector into the locations in each group, the failure transitions between locations in a group, and the transitions from each location to the following group connector using the $goals$, $elab$, $trans$, and usv inputs, as well as the location and constraints output from the constraint merging algorithm. The "starts in" logic for each goal in each location are combined using a logical 'AND' connector, and then simplified and checked for viability. If the resulting logical expression evaluates to False, the transition is removed. The transition logics for all of the constrained state variables based on the constraint types present in the location are also combined and simplified. Failure logic for each goal in each location is listed, and the conditions for failure transitions to the same location are combined with a logical 'OR' connector and are simplified. The locations that are receiving these failure conditions are updated to reflect that fact. An unsimplified $cdsva$ structure is the output of this algorithm.

The location removal algorithm checks several conditions that would warrant the removal of the location or group for each location and group and then removes the location or group if any condition is true. These conditions include the lack of input transitions, all locations having invariantly True exit conditions, an invariantly True failure condition, or exit conditions that are a subset of the conditions of all incoming transitions. The removal of the locations include the removal of all other failure transitions originating from that location, and the reassignment of accepting locations for any entry transitions into the deleted location. The output of this algorithm is the final output of the software, the $cdsva$ list.

IV. Benchmark Problems

A. Mars Rover Example

This example was first introduced in a previous work [14]. It involves a robot with three sensors traversing a path to get to one point of interest, whose selection depends on the health of the sensors. Figure 3 depicts the goal network and the hybrid system to which the goal network is converted. The inputs to the conversion software consist of eleven inputs in the $goals$ list; two controllable state variables, position with four constraint types, and orientation with two constraint types; one uncontrollable state variable, system health, with three discrete states; and six parent goals with elaboration logic. The output of the conversion software, shown in Figure 4, matches the manual version of the converted hybrid automaton exactly and was found in less than one second using the software.

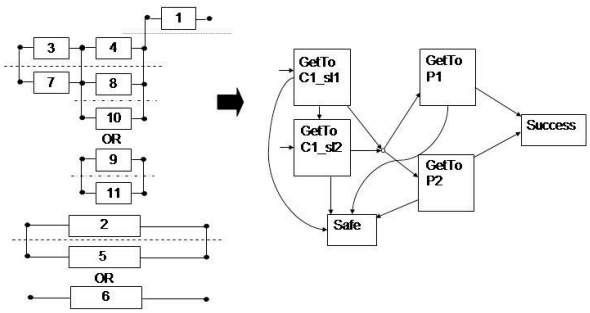

Fig. 3. Numbered goal network and its hybrid automaton conversion for the Mars rover example

1.1) GetToC1_sl1: {2,3,5,7}, {{1,4,{6,10}},{2,1,{0}}}, {GOOD==SH}, {v<=10, x==6&&theta==0}, {{FAIR==SH, 2},{POOR==SH, Safe}}, {}

1.2) GetToC1_sl2: {2,3,6,7}, {{1,4,{6,5}}, {2,1,{0}}}, {FAIR==SH}, {v<=5, x==6&&theta==0}, {{POOR==SH, Safe}}, {{FAIR==SH, 1}}

2.1) GetToP1: {2,4,5,8,10}, {{1,4,{4,10}}, {2,1,{30}}}, {GOOD==SH}, {v<=10, x==4&&theta==30}, {{FAIR==SH, 2}, {POOR==SH, Safe}}, {}

2.2) GetToP2: {2,4,6,9,11}, {{1,4,{4,5}}, {2,1,{-30}}}, {FAIR==SH}, {v<=5, x==4&&theta==-30}, {{POOR==SH, Safe}}, {{FAIR==SH, 1}}

Fig. 4. Annotated conversion software output for the Mars rover example

B. Where's Waldo Example

The Where's Waldo example problem was taken from another work that solves the problem using a correct by design control system that is generated from requirements written in a restricted set of LTL [10]. A known rectangular course broken into four regions (shown in Figure 5) is traversed by a robot that looks for Waldo, who can be in region P2 or P4. If the robot finds Waldo in one of these locations, the robot must stay there; otherwise it must continue to look. A segment of goal network used to solve this problem is found in Figure 6 along with the hybrid automaton that the conversion software found. The input to the software consisted of ten goals; one controllable state variable, position, which had two constraint types (drive and maintain); one uncontrollable state variable, the location of Waldo, which had three discrete states (*P2*, *P4*, or $NotPresent$); and four parent goals with elaboration logic.

The hybrid automaton specified by the conversion software was run in PHAVer with an unsafe (or incorrect) set that included staying in P2 or P4 when Waldo was not present and leaving P2 or P4 when Waldo was present in the same region as the rover. The model checker found no executions of the hybrid automata that would cause the robot to enter the unsafe set. The solution for this problem was found and verified very simply using this goal network conversion and verification procedure.

C. Complex Goal Network Example

A contrived example to test the capabilities of the conversion software was developed. This example consists of a goal

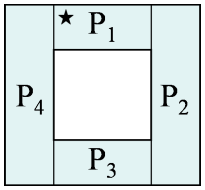

Fig. 5. Environment for the Where's Waldo example. The star indicates the initial rover position [10].

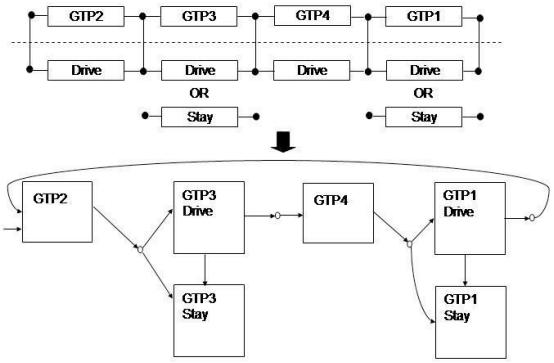

Fig. 6. Goal network and its hybrid automaton conversion for the Where's Waldo example, where GTP2 = GetToP2, etc.

network with twenty-five goals; two controllable state variables, the first with four constraint types and the second with three constraint types; two uncontrollable state variables, each with three discrete states; and nine parent goals with corresponding elaboration logic. A graphical representation of the goal network and resulting hybrid automaton are shown in Figure 7. The conversion software successfully found the correct automaton in about one second.

V. Conclusions and Future Work

The goal network conversion software presented in this paper is capable of quickly and accurately converting complicated goal networks into linear hybrid automata that can be verified using existing symbolic model checking software such as PHAVer. This automatic tool allows more goal networks to be verified due to its speed and ease of use, which is a promising development for goal-based control programs, particularly for control architectures such as MDS.

There is much work to be done with this software. Many capabilities covered in the original conversion process can be added to this software. These include dynamical equations for each controllable and dependent state variable constraint type, model representations of the evolution of the uncontrollable state variables, time constraints, replan states, and removal of unnecessary cycles of locations. Another capability that will be added soon is a parser to translate the output $cdsva$ into PHAVer code. Another parser that can take a more intuitive goal network definition and translate it into the input lists would also be very useful.

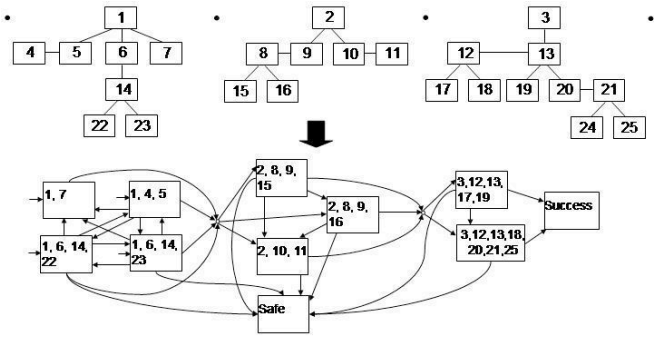

Fig. 7. Numbered goal network elaboration tree and its hybrid automaton conversion for the complex goal network example

VI. Acknowledgements

The authors would like to gratefully acknowledge Kenny Meyer, Michel Ingham, David Wagner, Robert Rasmussen, Kirk Reinholtz, and the rest of the MDS team at JPL for feedback, suggestions, answered questions, and MDS and State Analysis instruction; and Hadas Kress-Gazit for the Where's Waldo example. This work was funded by AFOSR.

References

[1] G. Labinaz, M. M. Bayoumi, and K. Rudie, "A survey of modeling and control of hybrid systems," *Annual Reviews of Control*, 1997.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

[12] 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.

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

[14] J. M. Braman, R. M. Murray, and D. A. Wagner, "Safety verification of a fault tolerant reconfigurable autonomous goal-based robotic control system," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

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