A Hybrid Systems Framework for Multi-robot Control and Programming

Joel M. Esposito
Vijay Kumar
GRASP Laboratory, University of Pennsylvania
jme,kumar@grip.cis.upenn.edu

April 26, 2002

Abstract

In this paper, we present a framework and the software architecture for the deployment of multiple autonomous robots in an unstructured and unknown environment with applications ranging from scouting and reconnaissance, to search and rescue and manipulation tasks. Our software framework provides the methodology and the tools that enable robots to exhibit deliberative and reactive behaviors in autonomous operation, to be reprogrammed by a human operator at run-time, and to increase system reliability in unstructured, dynamic environments.

1 Introduction and motivation

There is extensive literature on the control of robot manipulators or mobile robots in structured environments, and traditional robot control is a well understood problem area. However, traditional control theory mostly enables the design of controllers in a single mode of operation, in which the task and the model of the system are fixed. The last few years have seen active research in the field of control and coordination for multiple mobile robots in unknown and unstructured environments, with applications including tasks such as exploration [2], surveillance [6], search and rescue [8], mapping [19, 7], distributed manipulation [16, 10] and transportation of large objects [17, 18]. A review of contemporary work in this area is presented in [11]. When operating in unstructured or dynamic environments with many different sources of uncertainty, it is very difficult if not impossible to design a single continuous controller that will guarantee performance even in a local sense. Any real world applications will require using several different controllers for different phases of the task, as well as using multiple estimation algorithms and even possibly multiple plant models. In addition, messaging over a local wireless network, geometric planning algorithms and image processing routines are constantly being run, adding a discrete and algorithmic component to control and estimation. In light of these issues, designing reliable systems is a challenging task especially as the number of robots gets large.

The challenge in designing such systems is that as the switching logic becomes more complex, the difficulty in maintaining and debugging the code increases dramatically unless good programming practice is used – and it is all too easy, in trying to quickly put together an application, to write code in haste. It is not uncommon for programmers to write blocks of code encased in complex erroneous conditional statements whose execution is precluded; and all too difficult to discover the sources of runtime errors. Furthermore, control laws, estimation schemes, plant models, and algorithmic components can become inextricably entangled, forcing one to "reinvent the wheel" each time a new application is created or when there is a change of platforms. Poorly organized code needs to be completely redesigned when changes must be made or new tasks are introduced. A deeper issue lies in predicting the aggregate behavior of the system when each piece of code is designed and tested in isolation. It is well known that mixed discrete/continuous system can exhibit behavior not predicted by either control theory or discrete mathematics alone. Therefore as robotic applications become increasingly sophisticated and safety critical, it is imperative that programmers have a formal way of designing, representing, and programming such software based control systems.

Our goal, in this paper, is to describe a set of software tools that allows the development of controllers and estimators for multi-robot coordination. The tools consist of a framework for developing software components, and an architecture for composing control and estimation modules. Our software framework divides the overall multi-robot control task into a set of modes or behaviors, which may be executed either sequentially or in parallel. Modes can consist of high-level behaviors such as planning a path to a goal position, as well as low-level tasks such as obstacle avoidance. We use a high-level language to formally describe how and when transitions between these modes are to take place in order to achieve a set of global objectives.

The benefits of using such formalisms include: ease of maintenance; rapid development of new applications; improved performance estimates; greater code reuse; and guaranteed extensibility. The remainder of the paper is organized as follows: Section 2 provides a mathematical definition of our prefered modeling formalism, Hybrid Automata. Section 3 illustrates how this definition can be applied to robot systems. Section 4 lists basic features of modern programming languages that facilitate efficient software development and maintanence. Section 5 explains how we have incorporated these concepts into the CHARON programming language and illustrates its main features through examples. Finally in Section 7 we explain how the formal model can be exploited for the purposes of simulation and, eventually, analysis.

2 Definitions

In this section we discuss our modeling formalism, *Hybrid Automata*, and illustrate how it can be used to model robot systems.

A hybrid system can be viewed in many ways: perhaps as a generalization of a switched control system, or as a finite state machine augmented with differential equations. Many modeling paradigms have been introduced (see [1] for an overview). A commonly accepted model is that of a Hybrid Automata. The definitions that follow

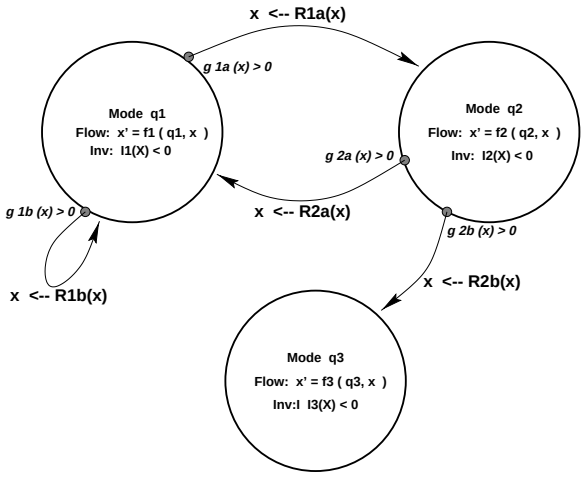

Figure 1: A graphical representation of a generic hybrid automata.

have been adapted from [9]

Hybrid Automata A hybrid automaton is a collection $H = (Q, X, Init, f, I, E, G, R)$, where

- Q is a countable set of discrete variables;
- X is a set of continuous variables;
- $Init \subset Q \times X$ is the set of initial states;
- $f : Q \times X \to TX$ is a vector field defining the continuous flow;
- $Inv : Q \to 2^X$ assigns to each $q \in Q$ an invariant set;
- $E \subset Q \times Q$ is a collection of possible discrete transitions (edges);
- $G : E \to 2^X$ assigns to each edge, $e = (q, q') \in E$, a guard; and
- $R : E \times X \to X$ assigns to each edge, $e = (q, q') \in E$, a reset relation.

This definition represents a fairly broad and powerful modeling framework which subsumes continuous differential equations, discrete time difference equations, finite automata, and timed automata; as well as switched control systems and non-smooth differential equations. A more intuitive explanation of the components of the Hybrid Automata definition is presented via an example in Section 3 Frequently, graphic representations of hybrid automata appear in the literature. They are depicted as shown in Figure 1.

The evolution of a hybrid automata's state over time is called an *execution* and is essentially a concatenation of continuous flows (like those of ordinary differential

equations), and discrete transitions (consisting of mode changes and resets). An execution begins at some state $(q, x) \in Init$ and the continuous state's evolution is governed by a differential equation prescribed by the flow $\dot{x} = f(q, x)$. Time increases during this part of the evolution. If at anytime during the continuous flow, there exists an edge $\{e = (q, p) : p \in Q\}$ connecting the current discrete state, q, to any other discrete state, p , for which the edges associated guard condition $g_e(x)$ is true, a transition will occur. When a transition occurs, two things happen: the value of the current discrete state jumps from q to p and a reset mapping associated with the edge $e = (q, p)$ is applied which changes the value of the continuous state. The reset mapping $R(e, x) \rightarrow x'$ may be a simple algebraic function or a very complex algorithm. This transition is assumed to occur *instantaneously* (*i.e.* no time flows during the discrete update). After the transition occurs, time resumes its passage and the continuous state now flows according to $\dot{x} = f(p, x)$ with initial condition x'. The process continues this way.

3 Modeling

The various sets and mappings which constitute the definition of a Hybrid Automata can be explained in a mobile robotics context. As an illustrative example, consider a security mobile robot with unicycle type dynamics whose mission is to traverse the perimeter of a warehouse while looking for intruders; if the robot accidentally collides with the fence or some unexpected obstacle, it should back up and try to resume its patrol; if an intruder is located the robot should notify the police via its radio link and follow the intruder until they arrive.

The robot and its controllers and estimators are modeled as a hybrid system (see Figure 2). Returning to the Hybrid Automata definition,

- Q is the set of possible modes, in this case each mode is a different controller and estimator for best suited for a different part of the task, they are labeled $q \in \{$*patrolling*, *tracking*, *collision recovery* $\}$.

- X is the set of continuous states in the traditional sense (*i.e.* position, velocity, etc.), the dimension of the state may change in each mode. For example, in the *tracking* mode, the intruder's position may be considered a state variable, while in the other modes, only the robot's position and velocity are part of the state.

- $Init$ is the set of initial conditions for the robot, this may be the entire state space in this situation.

- $f(q, x)$ $q \in Q$, and $x \in X$, is the continuous dynamics of the closed loop system, as one would expect f is not necessarily a continuous function of the discrete state q; although, if q is held constant, f is continuous in x. The controller used in *patrolling* and *tracking* may be qualitatively different closed loop controllers; while the controller used in *collision recovery* is simply an open loop controller which causes the vehicle to travel in reverse for a brief period.

- Inv, is a condition the designer believes will remain true when a given mode is active. If this condition is violated, a flag can be raised, a message sent or

corrective action taken. This tool is primarily diagnostic. For example, during the *patrol* mode, one may want to verify they the robot is never greater that 5 meters from the building. If at anytime the controller overshoots this amount, a warning is logged.

- E captures the connectivity of the modes. E is the set of all possible mode transitions $\{patrol \rightarrow tracking, patrol \rightarrow collsion, tracking \rightarrow patrol, \dots\}$. It is even possible to have "self-transitions" $\{tracking \rightarrow tracking\}$.

- G is the set of logical conditions which determine when mode switches occur. For example the transition $patroling \rightarrow tracking$ may occur when $intruder = true$ mathematically, $intruder$ is a inequality condition on the sensor readings.

- R is an algebraic assignment or algorithm which is executed upon transition. For example when the transition is enabled $patroling \rightarrow tracking$, a wireless message might be sent to the police or to the human security guards; or for example upon colliding with an obstacle the robot's velocity might be set to zero.

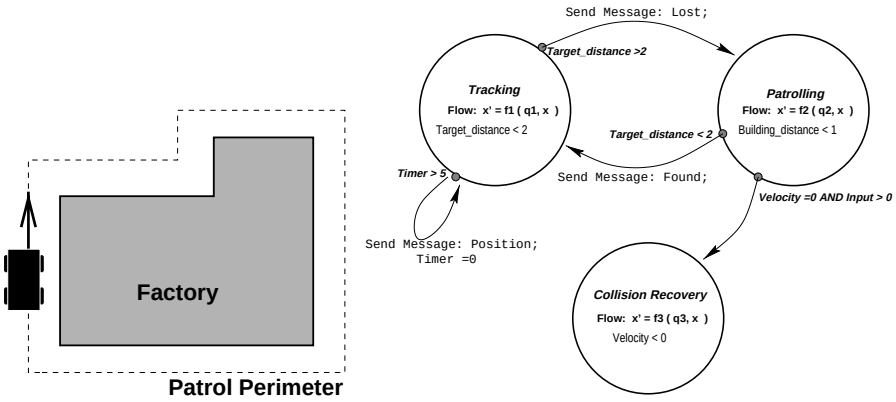

Figure 2: *Left* The security patrol application. *Right* The robot's behavioral program modeled as a hybrid automata.

While the previous example illustrates one way the Hybrid Automata framework can be used to model a software based control system, there are clearly many modeling decisions to be made when applying the framework.

4 Software design concepts

One also needs to keep in mind that the Hybrid Automata framework is simply an abstract mathematical entity. There are many ways to represent this entity in software. Given a choice of representations one would like to use a representation which facilitates efficient programming, ease of maintenance, etc. Ideally a representation would

capitalize on the desirable features of other modern programming languages such as: *formalism*; *modularity*; and *hierarchy*.

The advantages of using a *formalism* are many. Adhering to a formal programming paradigm ensures consistency and readability of the code, which is critical when projects involve more than one programmer. Concepts like type checking and scoping encourage better programming practice, and throw compile time errors when poorly written code is used. Finally, having a formal language means that a uniform graphical representation, such as the one in Figure 2 can be used so that programmers can visualize the final design.

Modularity implies that various pieces of code can be interchanged with minimal effort. For example if one would like to replace the estimator used in the *tracking* mode with a different estimation scheme one should not have to rewrite the entire closed loop dynamics of the mode but rather just simply swap the part of the code that contains the estimation scheme. In order to do this one needs to write code in which the distinction between the dynamics, estimator and controllers are clearly defined, in much the same way that it is desirable when programming other types of applications to break large tasks into subroutines and procedures rather than write one large block of code. Clean interfaces must also be used, so that it is clear what the "inputs" and "outputs" of any new block of code must be to ensure that it is compatible with the existing software. The advantages of this approach are that it is easy to maintain code, by simply updating parts of it; and quicker to develop new applications because old blocks of old code can be reused. In a similar fashion, the use of *hierarchy* also reduces development times because it limits the amount of code that needs to be rewritten. It allows more complex programs to be built up out of simpler blocks of code.

5 The CHARON Language

We have developed CHARON, an acronym for Coordinated Control, Hierarchical Design, Analysis, and Run-Time Monitoring of Hybrid Systems, a high-level language to facilitate the programming of multiple, interacting hybrid systems. The language is designed with the goal of being able to control multiple mobile, autonomous robots for mission-critical applications and stringent requirements on safety. More details about the language, the semantics and the formal description are presented in [14], [13]. Here we will illustrate the language's features informally through the use of examples. Other examples appear in [15] and [12] and on the CHARON web page

`http://www.cis.upenn.edu/mobies/charon/`

5.1 Formalism

Formal CHARON code for the mode *patrolling* could be written as follows. Lines with `//` are comments, which explain the purpose of each line to the reader.

```
mode patrolling(real v_max, real k)  {
// above line names the mode class and describes the
// parameters that must be specified when it is created
```

```
// (in this case: maximum forward speed v_max
// and a gain k.

    // this is the name of its entry point, it is used in
    // describing the transition edges later
    entry enterPatrol

    // the name of the exit point
    exit  exitPatrol

    // these are the continuous (called analog) state
    // variables  (position and orientation)
    // along with the reference steering angle phi.
    // they are readWrite since they will be both read
    // to evaluate various functions within the mode,
    // as well as written to as time flows during
    // the continuous evolution
    //
    // Note that externally defined types such
    // as ‘‘Position’’ can be used
    readWrite analog Position myPos;
    readWrite analog real theta;
    readWrite analog real phi;

    // this defines the differential equations for the
    // continuous states
    diff diffSteer { d(myPos.x) == v*Math.cos(theta);
                     d(myPos.y) == v*Math.sin(theta);
                     d(theta) == omega }

    //  these are algebraic assignments.  In one case a
    //  simple control law in the other an external
    //  planning algorithm is called to compute the
    //  desired reference signal phi
    alge algePhi  phi == externPlanner(myPos, theta)
    alge algeOmega  omega == k * (theta - phi)
    alge algeV  v == v_max

    // the invariant in this situation is simply that
    // the robot’s distance to the
    // building is less than some threshold
    inv invBuildingDistance externDistToBuilding <= 5

}       // end of mode patrolling
```

First note that this is essentially a *class* definition in the object oriented programming sense – it is a template. Many instances of the mode class can be defined. For example `aggressivePatroling` may be defined as `patrolling( vmax =5, k = 10)` while `slowPatrolling = patroling(vmax =1, k = 2)`.

In general `diff` is used to define the flow f, `inv` denotes invariants Inv, and `alg` is used to define simple algebraic assignments (such as control laws). Variables, are typed as real, int, etc., and externally defined types are also allowed as in the case of `position`. In addition to typing, variables can be discrete or analog. Analog variables are updated continuously, while discrete variables are updated only upon initialization and mode switches. Likewise, one could write specifications for the remaining modes

```
mode tracking(real v, position target, real k) {
    ⋮
}       // end of mode

mode collisionRecovery(real v) {
    ⋮
}       // end of mode
```

Note that the entry and exit points for a mode are labeled using the key words `entry` and `exit`, however the guards and reset maps are not defined within the individual modes since they describe how the modes are connected (a global property) rather than being local properties of the individual modes. The "top level" mode is a mode which contains such global information. The top level mode also creates (or instantiates) particular instances of the individual modes, specifying values of their parameters.

```
mode topRobotMode(Position initPos,
Position target, real v, real phi, real initTheta, real initOmega, real k)

    entry entryPt;

    //  defining variables
    readWrite analog Position myPos;
    private analog real timer;

    diff diffTime d(timer) == 1.0

    //  instantiate the three modes
    mode slowPatroling = patrolling(2, 2)
    mode aggressiveTracking = tracking(4, target, 5)
    mode backup = collisionRecovery(-2)

    // define all the transitions (guards, edges and resets) connecting the mode
    trans from slowPatrolling.enterPatrol to backup.entry when (collide==true) d
        myVel = 0.0; timer = 0.0 }
```

```
        ⋮
   trans arrival from backup.exit to slowPatrolling.enterPatrol
        when (timer==2) do {}
        ⋮
 // end of mode top
```

5.2 Modularity

One type of modularity was illustrated in the specification of the `topRobotMode` above. Many different individual modes can be designed in isolation and composed as needed by instantiating them and specifying the edges between them. This is often referred to as *sequential composition*. Another type of composition – *parrelel composition* – is described below by introducing the concept of an *agent*.

An agent is simply a software entity defined by grouping parts of the model together under a top mode for convenience and specifying some variables as `input` and other as `output`. The input-output dependencies define an interface and let one compose various agents to get a larger system. In CHARON the `channel` keyword is used to create such an interface. For example in a mobile manipulator system where x is the set of state variables of the mobile base and y is the set of state variables for the manipulator, the system dynamics are

$$\dot{x} = f_b(x, y) \quad (1)$$
$$\dot{y} = f_m(x, y), \quad (2)$$

a single CHARON agent can be written for the system

```
agentRobot(...)  mode top =  topRobotMode( ...)
```

where the top mode is

```
mode topRobotMode() {
    ⋮
  private analog real x, y;
  diff dynamics{d(x) ==  f_b (x,y) ;
                d(y) ==  f_m (x,y) ; }
    ⋮ }
```

However, it may be more advantageous to define the base and manipulator dynamics as separate agents and compose them.

```
agent base(){mode topBase=topBaseMode()}
agent manipulator(){mode topManipulator=topManipulatorMode()}
```

with the top mode defined as

```
mode topBase() {
    ⋮
  channel of real myState
  channel of real otherState
  receive (otherState, y)
  send (myState, x)
  diff dynamics{d(x) ==  f_b (x, y)  }
    ⋮ }


mode topManipulator() {
    ⋮
  channel of real myState
  channel of real otherState
  receive (otherState, x)
  send (myState, y)
  diff dynamicsd(y) ==  f_m (x, y) ;
    ⋮ }


agent Robot() {

  private channel of real baseToManipulator,
                                        manipulatorToBase;
  agent theBase = base () [baseToManipulator,
                                        manipulatorToBase]
  agent theManipulator = manipulator () [baseToManipulator,
                                         manipulatorToBase]

}
```

The key is that the two are different ways of representing the same Hybrid Automata – mathematically they are identical. It is only their software representation that is different. But the second representation may be much more useful because individual model components can be redefined as needed and composed into a new system.

In general inter-connected agents behave like block diagrams in terms of signal flow. Thus controller agents, plant agents, and estimator agents can be defined and wired together using the `channel` keyword. Another application of agent composition is to convey discrete signals between two systems rather then to express continuous signal flow. For example many copies to the robot agents can be instantiated and a wireless network can be modeled using channels. The resulting system is a team of robots.

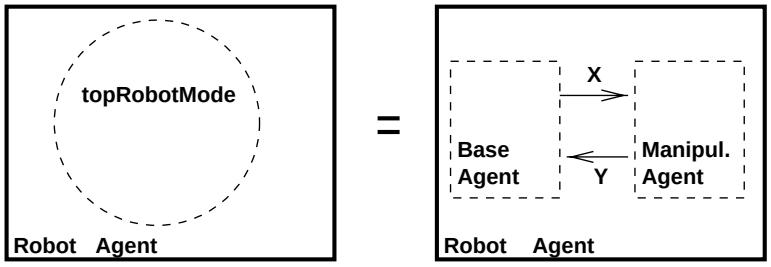

Figure 3: A graphical representation of parallel composition of the base and manipulator agents. From an input-output point of view the two representations are externally identical.

5.3 Hierarchy

One type of hierarchy was illustrated in the previous section. A manipulator agent and a base agent were composed together as a robot agent. In such a situation where mode complex agents are built out of less complex agents we refer to the agents as super-agents and sub-agents respectively. This is often called architectural hierarchy.

Another type of hierarchy occurs at the mode level where, likewise, sub-modes and super-modes are used. This is called behavioral hierarchy. A two level hierarchy was seen in the patrolling example when the basic (or atomic) modes *patrolling*, *tracking*, and *collisionRecovery* were grouped together inside the *topMode*. However, a behavioral hierarchy can have any number of levels. Suppose we wanted to program a general purpose robot, capable of executing many different missions. We might rename what was originally referred to as *topMode* to *securitySuperMode*. Then we might reuse some of the *patrolling*, *tracking*, or *collisionRecovery* modes along with newly designed atomic modes *exploration*, *pushing*, *goToGoal*, etc. to build other high level behaviors in addition to *securitySuperMode* as in Figure 4. High level modes such as *surveyingSuperMode*, *searchAndRescueSuperMode*, etc. could be constructed. Even higher level behaviors could be constructed out of these super modes.

6 Implementation

The CHARON compiler and simulator was written in Java. However the low-level implementation of this programming structure used to control actual hardware platforms was done in C++. It uses *Live Objects*. *Live Objects* have been developed as part of the software architecture for implementation on the hardware platforms. A live object encapsulates algorithms and data in the usual object-oriented manner together with control of a thread within which the algorithms will execute, and a number of events that allow communication with other live objects. At the top of the hierarchy, the algorithms associated with the objects are likely to be planners, while at bottom they will be interfaces to control and sensing hardware. The planner objects are able to control the execution of the lower level objects to service high-level goals. To offer platform independence, only the lowest level objects should be specific to any hardware, and these

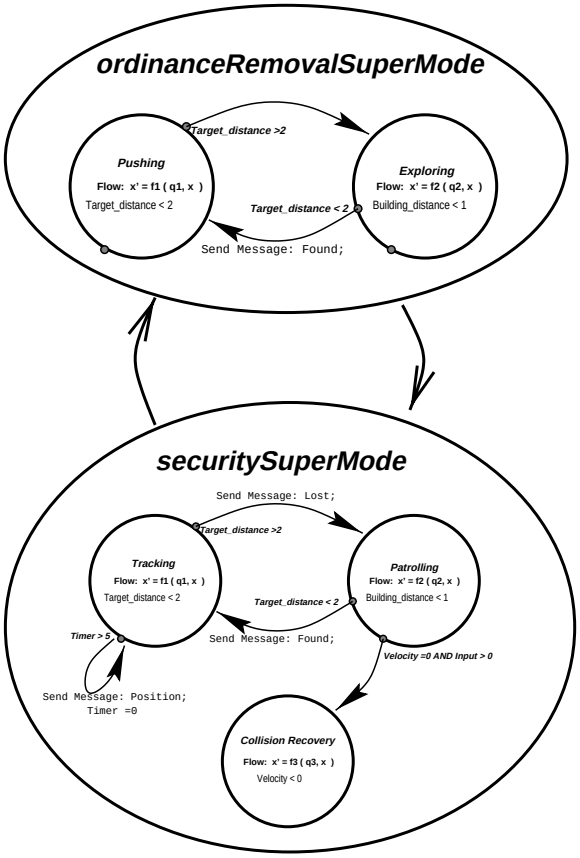

Figure 4: A graphical representation of hierarchical composition of modes to form higher level behaviors.

should have a consistent interface for communication with the more high level planning objects that control their execution. Visual servo control algorithms have been incorporated into the live object framework for such basic functionality as obstacle avoidance, wall-following, formation keeping, mapping and localization.

7 Discussion

The principle advantage of modeling systems using a formal framework is that the structure of the model is preserved and can then be exploited for other purposes. Throughout this tutorial we have tried to demonstrate how this structure could be exploited to reduce software development and maintenance times. We showed how, using hierarchy and modularity, new applications can be prototyped quickly by leveraging and reusing existing software modules.

We are currently working toward exploiting this structure for producing more accurate and efficient simulations of Hybrid Automata. For example in [5], we develop a method of selecting integration step sizes when a discrete transition is about to occur in such a way as to ensure that the transition is properly detected and handled. The step size selection methodology uses concepts from nonlinear control systems design to correctly simulate system with model singularities, an area in which many other simulation algorithms fail.

Traditional simulators can be notoriously slow when applied to complex hierarchical or multi-agent examples. In [4] we address the simulation of multi-agent systems. We introduce the first algorithm which is capable of simulating, for example, a multi-robot team in an asynchronous way. It allows a different integration step size to be used for each agent rather than constraining the simulation to be executed with a single global step size. The algorithm produces significant efficiency gains by allowing each agent to use an individually selected, largest acceptable step size – yet it still properly handles discrete events which depend on the configuration of the overall team. Likewise, in [3] we address asynchronous methods for simulating hierarchical systems with similar results. Ultimately, one would like to exploit this structure for performing automated analysis such as verification, abstraction, as well as control synthesis. Some initial steps have been taken in this directions but significant challenges remain.

Acknowledgments

We gratefully acknowledge support from DARPA grant MOBIES F33615-00-C1707 and NSF CDS-97-03220. The first author is also partially supported by a DoE GAANN grant.

References

[1] Michael Branicky. *Studies in Hybrid Systems: Modeling, Analysis and Control*. PhD thesis, MIT, Cambridge, MA, 1995.

[2] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative multi-robot exploration. In *Proc. IEEE Int. Conf. Robot. Automat.*, pages 476–481, San Francisco, CA, April 2000.

[3] J. Esposito and V. Kumar. Efficient dynamic simulation of robotic systems with hierarchy. In *IEEE International Conference on Robotics and Automation*, pages 2818–2823, May 2001.

[4] Joel Esposito, George Papas, and Vijay Kumar. Multi-agent hybrid system simulation. IEEE Conference on Decision and Control, December 2001.

[5] Joel M. Esposito, Vijay Kumar, and George J. Pappas. Accurate event detection for simulating hybrid systems. In M.D. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems : Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 204–217. Springer Verlag, 2001.

[6] J. Feddema and D. Schoenwald. Decentralized control of cooperative robotic vehicles. In *Proc. SPIE Vol. 4364, Aerosense*, Orlando, Florida, April 2001.

[7] L. Iochhi, K. Konolige, and M. Bayracharya. A framework and architecture for multi-robot coordination. In *Proc. Seventh Int. Symposium on Experimental Robotics (ISER)*, Honolulu, Hawaii, Dec. 2000.

[8] J. S. Jennings, G. Whelan, and W. F. Evans. Cooperative search and rescue with a team of mobile robots. *Proc. IEEE Int. Conf. on Advanced Robotics*, 1997.

[9] John Lygeros and George Pappas. A tutorial on hybrid systems: Modeling, analysis and control. 14th IEEE International Symposium on Intelligent Control/Intelligent Systems and Semiotics, September 1999.

[10] M. Mataric, M. Nilsson, and K. Simsarian. Cooperative multi-robot box pushing. In *IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pages 556–561, Pittsburgh, PA, Aug 1995.

[11] L. E. Parker. Current state of the art in distributed autonomous mobile robotics. In L. E. Parker, G. Bekey, and J. Barhen, editors, *Distributed Autonomous Robotic Systems*, volume 4, pages 3–12. Springer, Tokyo, 2000.

[12] R.Alur, C.Belta, F.Ivancic, V.Kumar, M.Mintz, G.Pappas H.Rubin, and J.Schug. Hybrid modeling of biomolecurlar networks. *Hybrid Systems Computation and Control*, 2001.

[13] R.Alur, R. Grosse, I. Lee, and O.Sokolsky. Refinement for hybrid systems in charon. *Hybrid Systems Computation and Control*, pages 12–21, 2001.

[14] R.Alur, R. Grosse, Y.Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. *Hybrid Systems Computation and Control: Third international workshop*, 3:6–19, 2000.

[15] R.Fierro, Y.Hur, I.Lee, and L.Sha. Modeling the simplex architecture using charon. In *Proceeding of the 21st IEEE Real Time Systems Symposium*, pages 77–80, 2000.

[16] D. Rus, B. Donald, and J. Jennings. Moving furniture with teams of autonomous robots. In *IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pages 235–242, Pittsburgh, PA, Aug 1995.

[17] D. Stilwell and J. Bay. Toward the development of a material transport system using swarms of ant-like robots. In *IEEE International Conf. on Robotics and Automation*, pages 766–771, Atlanta, GA, May 1993.

[18] T. Sugar and V. Kumar. Control and coordination of multiple mobile robots in manipulation and material handling tasks. In P. Corke and J. Trevelyan, editors, *Experimental Robotics VI: Lecture Notes in Control and Information Sciences*, volume 250, pages 15–24. Springer-Verlag, 2000.

[19] C. J. Taylor. Videoplus: A method for capturing the structure and appearance of immersive environment. *Second Workshop on 3D Structure from Multiple Images of Large-scale Environments*, 2000.