Formal Synthesis of Supervisory Control Software for Multiple Robot Systems

J. Goryca and R. C. Hill

Abstract— **This paper demonstrates the application of a range of theoretical tools to generate real-time control software for multiple ground robots working together cooperatively. Specifically, existing discrete event system theory is applied to synthesize high-level supervisory control logic that is guaranteed to maintain the behavior of multiple robots within requirements defined by a set of formal specifications. The modeling of the high-level behavior of the robots in their given environment, as well as the formal specifications, is described in detail. The resulting models are represented as finite-state automata. In this work we assume that some events cannot be controlled, though all events are assumed to be observable. In addition to generating control logic that is guaranteed to keep the robots *safe*, results are also presented for choosing from amongst a set of allowed robot behaviors in order to achieve behavior that is "good" in some sense. Specifically, a modified version of Dijkstra's algorithm is employed to choose a path through the finite-state automaton representing the allowed robot behaviors. This modified algorithm is able to address multiple robots and the fact that some events cannot be controlled (commanded). The resulting high-level robot events are then connected to the continuous, time-driven behavior of the robots through a series of low-level algorithms. The result of this work is demonstrated in simulation for a simple, but demonstrative scenario.**

I. INTRODUCTION

The cooperative control of multiple robot systems is an important research area with many practical applications ranging from perimeter security and surveillance, to search and rescue and firefighting [1]. The problem of cooperative robot control is also very challenging in that it often requires interaction with uncertain, unstructured environments, that provided goals and conditions can change abruptly, and that the complexity of the required control algorithms grow quickly as the number of agents increases.

Historically, the high-level control of complex systems has been generated in a heuristic manner based on designer understanding and intuition, or based on exhaustive simulation studies. This is true of vehicle control systems, as well as other domains such as manufacturing and computer system applications. The challenge here is that the process is generally time-consuming and can be prone to error, especially as the complexity of the systems increases.

The pressures of these challenges have spurred research in the area of formal synthesis of discrete control logic. The techniques that have been developed owe much to work from the computer science community on formally verifying the correctness of hardware and software [2]. These new synthesis techniques, however, don't just verify that some given logic satisfies a set of specifications; they automatically synthesize the logic to meet the given requirements by construction. Such techniques are valuable because they generate logic that is provably correct without need of exhaustive testing. Furthermore, the logic can be generated quickly upon changes in requirements or the environment.

One class of formal synthesis techniques is sometimes referred to as *reactive synthesis* [3], [4], [5], [6]. Many of these and related works specify the desired controlled behavior using process algebras (predicate calculus logic) such as computational tree logic (CTL) and linear temporal logic (LTL). Such formalisms provide a limited "language" of atomic propositions from which to specify the desired behavior. These formalisms can express notions of *safety* and *liveness*. It is argued that such process algebras are advantageous since they resemble a natural language description of specifications. However, most computational work is performed by converting such logic into Büchi automata. Büchi automata are a formalism that extend finite-state automata to accept infinite inputs, in particular, ω-regular languages [7].

The framework that will be employed in this paper is the supervisory control framework initiated by Ramadge and Wonham [8]. This type of discrete event control differs from those works mentioned so far in that it primarily relies on finite-state automata models and is able to address events that may not be controllable and events that may not be observable. (Though, work does exist where this formalism has been extended to the control of infinite behaviors [9].) Such a framework has great promise in that it can directly address uncontrolled events from the environment or a human user, as well events that cannot be observed either due to a sensor failure or lack of communication. Some work has been done to apply such discrete event techniques to the high-level control of autonomous agents, however, most of these cases are very simple scenarios involving one or two agents that do not address finding the optimal sequence of events from a set of allowed behaviors [10], [11].

In this paper we develop the models and apply the supervisory control framework to a relatively simple, but illustrative scenario involving the cooperative control of two robots. The purpose of this work is to begin to develop the tools and theory necessary to address more complex scenarios. In this paper we will assume that all events are observable, but that some events cannot be controlled. The product of the application of this supervisory control theory will be a finite-state automaton representing the set of all behaviors of the two robots allowed by the given set of requirements. A second aspect of this work is the implementation of an algorithm to choose a sequence of events from the allowed

All authors are with the University of Detroit Mercy, Detroit, MI 48221-3038, USA. Send correspondence to `hillrc@udmercy.edu`.

set of behaviors that is by some measure, "good." In this case, we will develop and apply a modified version of Dijkstra's algorithm that is able to optimize the behavior of a system of two robots that are operating concurrently. Additionally, our algorithm will take into account uncontrollable events.

The remainder of this paper is structured according to the following outline: Section II introduces supervisory control notation, Section III develops the models for our simple example, Section IV describes the structure and generation of the real-time control software, Section V describes the optimization algorithm developed for commanding the robots, Section VI presents some results from simulation, and Section VII concludes the paper.

II. Notation and Preliminaries

In this paper we will formally model the robots and their specifications as discrete event systems each represented by their own finite-state automaton $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is the set of states, Σ is the set of events, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, $q_0 \in Q$ is the initial state, and $Q_m \subseteq Q$ is the set of marked states representing the completion of a task. In general, the function δ is a partial function on its domain, where the notation $\delta(q, \sigma)!$ for any $q \in Q$ and any $\sigma \in \Sigma$ denotes that $\delta(q, \sigma)$ is defined. In this work we will assume that the automata are deterministic. This means that there is a single initial state and knowledge of which events have occurred completely determines the current state of the automaton. In other words, the same string of events may not lead the system to multiple different states.

The supervisory control framework employs a feedback-type architecture where the controller S makes decisions as to whether or not to allow the plant G to perform a given event based on which events have so far been observed. This paradigm is somewhat different than some other conceptions of "control" in that the supervisory controller does not command actions to take place, but rather, acts on top of the system to prevent any actions which will lead to a violation of the given set of specifications. The logic for choosing from amongst a set of allowed events will be described in Section V. In this paper we will represent the supervisory controller also as an automaton.

As the individual robots (and requirements) are represented by individual automata, the synchronous operation of the robots (and controller) will be captured employing the *synchronous composition* (or parallel composition) operator denoted $\|$. With this operator, an event that is common to multiple automata can only occur if that event is able to occur synchronously in each of the automata that share the given event. For example, a robot is able to perform an event only if that event is able to be generated by the supervisory controller automaton at the same time. If a component automaton employs an event that is not shared with any other automata, it may then enact the event without participation of the other automata. A formal definition for the synchronous composition of two automata is given below.

Definition 1: The *synchronous composition* of two automata G_1 and G_2, where $G_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, Q_{m1})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, Q_{m2})$ is the automaton

$$G_1 \| G_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{01}, q_{02}), Q_{m1} \times Q_{m2})$$

where the transition function $\delta : (Q_1 \times Q_2) \times (\Sigma_1 \cup \Sigma_2) \rightarrow (Q_1 \times Q_2)$ is defined for $q_1 \in Q_1, q_2 \in Q_2$ and $\sigma \in (\Sigma_1 \cup \Sigma_2)$ as:

$$\delta((q_1, q_2), \sigma) :=$$
$$\begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \delta_2(q_2, \sigma)! \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \sigma \notin \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \sigma \notin \Sigma_1 \text{ and } \delta_2(q_2, \sigma)! \\ \text{undefined} & \text{otherwise.} \end{cases}$$

As mentioned above, if the plant automaton has a feasible event at some instant of time that cannot be generated by the supervisory controller automaton at that instant, then that event is in effect disabled by the supervisor. Recall, however, that in this framework some events of the system are not controllable, that is, they cannot be disabled by the supervisor. For example, an action from the environment, another agent or a human user, may be uncontrollable. Such events need to be accounted for in the synthesis of the supervisor. Specifically, if an uncontrollable event will cause a violation of the system specifications, then the plant must be prevented (by disabling controllable events) from reaching the point at which that uncontrollable event is feasible. The supervisory control framework includes well-defined conditions and algorithms for synthesizing an "optimal" supervisory controller that will allow the supremal set of behaviors for which the plant is guaranteed to not violate a given set of formal specifications. The two classes of requirements that are captured by the supervisory control framework are *safety* and *nonblocking*. Safety means that the system will not perform any actions prohibited by a given set of specifications. In other words, the system will not enact a sequence of events that cannot be generated by the specification automata. Nonblocking means that the system can always reach a marked (goal) state. The reader is referred to other references for further details regarding the synthesis of supervisory controllers [12].

III. System Models

In this section we describe the models for a simple example that we will employ for demonstrating our approach to generating provably correct control software for multiple robot systems. In this scenario we will consider that we have two agents (robot A and robot B) that between them must complete a total of four tasks. In this example the tasks simply correspond to physical locations, but it could be envisioned that the robots must do something at each of the locations. The requirements of this scenario are that task 1 must be performed before task 2 and by the same robot, and task 3 must be performed before task 4 and by the same robot. A requirement like this could be necessary if it were required that a robot pick up an object at one location and then deliver that object to a second location. Another requirement is that the two robots are not allowed to be in the same quadrant of the field at the same time. Such a requirement might be added to prevent robot collisions, or to minimize the risk of losing both robots to an enemy agent. The overall scenario is depicted in Fig. 1 where the four quadrants are numbered from five to eight.

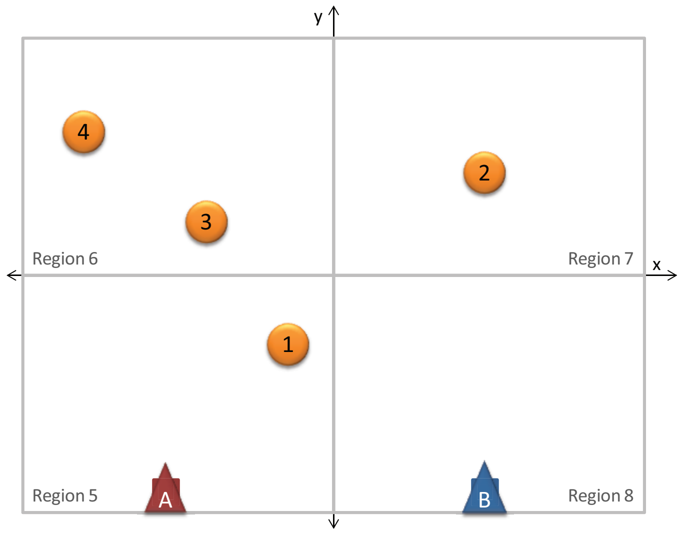
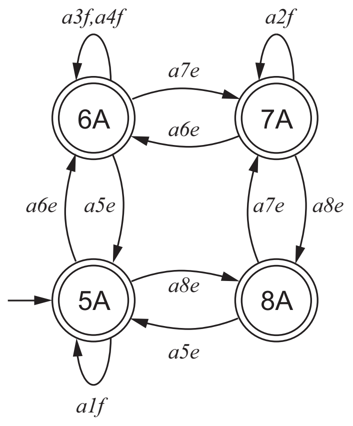


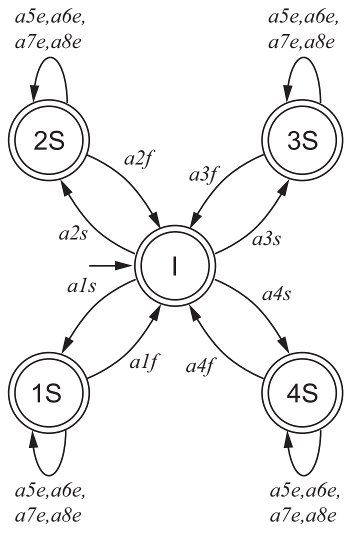
Fig. 1. Illustration of simple motivating example

The automata model of the robot A is generated from the synchronous composition of the two automata in Fig. 2. These automata reflect the possible behaviors of robot A under the constraints of the given environment without the addition of any supervision.

The event set for robot A is partitioned into sets of controllable ($\Sigma_{A,c} = \{a1s, a2s, a3s, a4s, a5e, a6e, a7e, a8e\}$) and uncontrollable events ($\Sigma_{A,uc} = \{a1f, a2f, a3f, a4f\}$). In this scenario, robot A can control the start of each task ($a1s$ = robot A starts task 1) and the entry into each region ($a5e$ = robot A enters region 5), but it cannot control when it finishes a task ($a1f$ = robot A finishes task 1). For example, once a robot has begun heading toward a task and crosses into the region where the task is located, then the robot can no longer be stopped. The automaton $G_{geog,A}$ in Fig. 2(a) captures the geographical constraints on robot A. Specifically, it captures the relative locations of the regions and the tasks, where the completion of a task corresponds to the robot's arrival at the corresponding location. This automaton also captures the initial location of robot A (region 5) where the initial state of the automaton is marked by the short arrow. Additionally, all states are marked (indicated by double circles) as the goal state will be defined by the specification automata. The automaton $G_{task,A}$ in Fig. 2(b) captures the fact that only one task can be performed at a time. In other words, a robot can only be headed towards one task location at a time. This automaton indicates that the robot is initially *idle* and will only move between regions as a consequence of trying to complete a task.



(a) Model of geographical constraints for robot A, $G_{geog,A}$



(b) Model of task constraints for robot A, $G_{task,A}$

Fig. 2. Models of the uncontrolled behavior of robot A as constrained by the environment

The synchronous composition of the two automata in Fig. 2 results in the automaton $G_A = G_{geog,A} \| G_{task,A}$ (not shown) which represents all possible behaviors of robot A without supervisory control. An analogous automaton model G_B exists for robot B, except that robot B has its own event set and begins in region 8. The composition of the automata representing robot A and robot B in essence represents the uncontrolled plant for this example. The composition $G = G_A \| G_B$ has 380 states and 1948 transitions. Since the models of robot A and robot B have completely disjoint event sets, their composition has a state space that is the full Cartesian product of the state spaces of the two component automata. This fact demonstrates the complexity challenge that arises in the synthesis of formal discrete event control. Specifically, as the number of component automata increases, the size of the full monolithic model can grow exponentially. Note that even though the individual robots can operate independently, their operation will be coupled by the control put in place to meet the specifications imposed on the system.

In addition to the formal plant models, we also need to generate models of the specifications. The automaton R_{avoid} in Fig. 3 represents the requirement that the two robots cannot be in the same region at the same time. This model can be generated from the composition of the two automata representing the geographical constraints on each of the robots (without finish events) with the "forbidden" states deleted. In other words, the states representing both robots in the same region $[(5A, 5B), (6A, 6B), (7A, 7B), (8A, 8B)]$ are deleted from the composition.

The automata in Fig. 4 capture the fact that task 1 needs to precede task 2 and be performed by the same robot (R_{12}), and task 3 needs to precede task 4 and be performed by the same robot (R_{34}). These automata also identify the "goal" for this scenario by marking only the states representing the completion of tasks 1 and 2 in the first automaton, and the completion of tasks 3 and 4 in the second automaton. The global specification can then be generated by the composition of the three individual specification automata, $R = R_{avoid} \| R_{12} \| R_{34}$. Marked states in the resulting composition will have the form $(*, 1f.2f, 3f.4f)$, representing the completion of all four tasks.

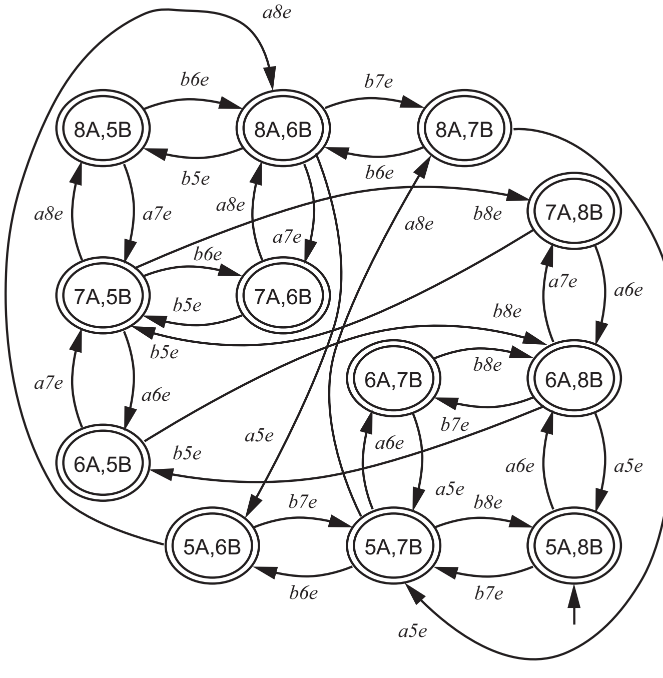


Fig. 3. Automaton model representing the requirement that the two robots cannot be in the same region at the same time, R_{avoid}

The composition of the plant model G with the specification model R represents the exact behavior that can be performed by the plant and is allowed by the specification. This automaton can represent the supervisory controller for our system. In essence, the controller automaton runs in parallel with the plant and updates its state as the plant generates events. The feasible event set of the controller automaton at any given state determines which events the plant is allowed to perform. The issue that arises, however,

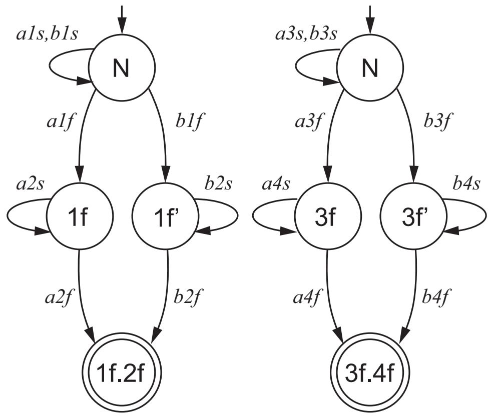


Fig. 4. Automata models representing requirements on the ordering and completion of the four tasks, R_{12} and R_{34}

is that the controller that would achieve this exact set of behaviors would have to disable uncontrollable finish events, which is not physically possible. Therefore, it is desirable to find the largest set of safe behaviors that can be achieved by disabling only controllable events (termed *controllability*). In other words, the controller must disable a controllable event such that the system doesn't reach a state from which it can enact an uncontrollable event that causes a violation of the given specifications. The theory for finding a controller which achieves the *supremal controllable* subbehavior is described in [12]. For this example, the automaton representing the largest set of controllable behaviors that meets the given specifications has 636 states and 1666 transitions. This automaton represents the supervisor for our system.

IV. Control Software Implementation

The automaton representing the supervisory controller for our example is generated off-line using the freely-available software package UMDES/DESUMA [13], though any number of other software tools could have been employed. Following the creation of this automaton representation of the controller, it is then necessary to use it to generate (with input from a user) real-time control software to actually operate the robots. This process entails two parts. The first part is that a mapping between the events of the automaton and what they mean in the real physical world must be generated. The second part is that a "planning" algorithm must be employed to choose which events the robots should enact. Recall, the controller automaton represents the supremal set of behaviors that can be controlled to satisfy the given specifications. Since it is likely that there are multiple sequences of events that are safe and lead the system to a marked (goal) state, the algorithm is necessary to choose the sequence of events that is by some measure "best." A diagram of the software generation process is depicted in Fig. 5.

The primary input to this process is a text file (extension .fsm) generated from UMDES that represents the controller automaton. A software tool has been developed that takes this text file and, with input from a user through a graphical user interface, is able to generate a data structure used by

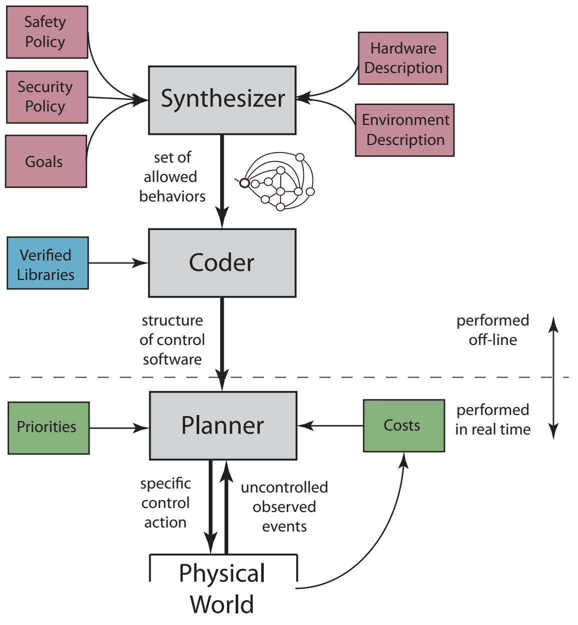


Fig. 5. Diagram representing the software generation process

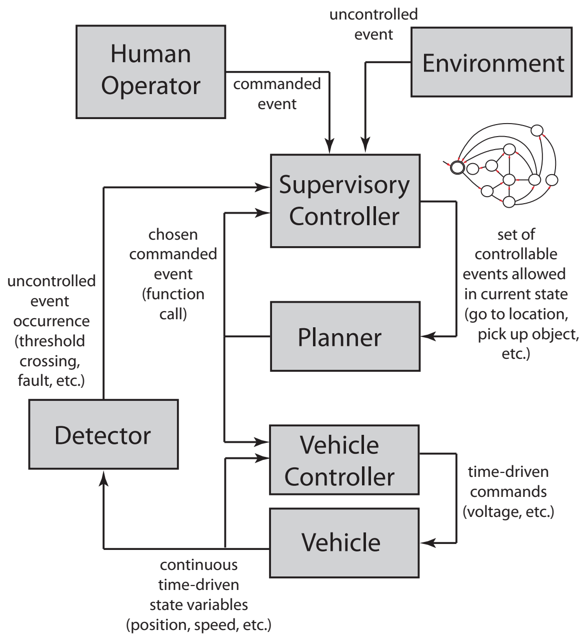

uncontrolled event
Human Operator
commanded event
Environment
Supervisory Controller
chosen commanded event (function call)
uncontrolled event occurrence (threshold crossing, fault, etc.)
set of controllable events allowed in current state (go to location, pick up object, etc.)
Planner
Detector
Vehicle Controller
time-driven commands (voltage, etc.)
Vehicle
continuous time-driven state variables (position, speed, etc.)

Fig. 6. Diagram representing the real-time control software architecture

the robots' real-time control software. The graphical user interface maps the events of the automaton to function calls of existing software modules. Specifically, controllable events map to function calls that command the robots to perform actions, while uncontrollable events map to software functions that detect the occurrence of events. For example, the controllable start event $a1s$ is mapped to a function call which sends robot A to task 1 using a series of lower-level algorithms. On the other hand, the uncontrollable finish event $a1f$ isn't "commanded," but rather is mapped to a function that simply detects when the task location has been reached. In this project, we specifically employ a D* Lite algorithm [14] to generate waypoints leading from the robot's starting location to its destination. We also employ a VFH* algorithm [15] to generate velocity and turn rate commands to send the robot to the intermediate waypoints while avoiding contact with obstacles within the range of the robot's laser. A mapping algorithm is also employed that localizes the robot's position on a map and provides the map as an input variable for the D* Lite and VFH* algorithms. Each of these algorithms have been validated and implemented on actual vehicles as part of prior research performed in the Advanced Mobility Laboratory at the University of Detroit Mercy. The diagram in Fig. 6 illustrates the overall software architecture employed in this project.

The data structure that is generated is then employed by a high-level control algorithm for each robot. The high-level control algorithm is written off-line and basically consists of a *while* loop that tracks the current state of the system and calls the relevant low-level functions as defined by the data structure. The element of the software that has not been discussed yet is how the system chooses between multiple controllable events when there are multiple legal actions available as defined by the supervisory controller automaton. This is discussed in the next section.

V. Task Planning Algorithm

The problem of choosing which action to take from amongst a set of legal events basically can be thought of as a graph search problem. We desire to find the path through the automaton representing the supervisory controller from the initial state to a marked state that is by some measure "best." In order to accomplish this, we must first assign costs to each of the events in our automaton. A variety of methodologies could be employed for assigning cost ranging from time between events, to probabilities related to the likelihood of an event occurring. In this example, we employed distances as our costs (which correlate to time). Specifically, each start event was assigned a cost equal to the straight-line distance between the robot's position at a given state and the location of the task. This facilitates implemention because we always know the robot's position preceding the enactment of a start event. This follows from the fact that every start event occurs either from a robot's initial position, or following the occurrence of a finish event which indicates the robot is positioned at the location of the task that was just finished. In this example, all entry and finish events were assigned costs of zero. This means that the high-level control decisions were based on the overall ordering of the tasks (1,2,3,4), not on the low-level paths that were actually taken to each location.

Once the costs are assigned, it is then necessary to find the lowest cost path through the graph. For this project, we employed a modified version of Dijkstra's algorithm for performing the optimization. For further details on the base algorithm, see [16]. The issue with the standard implementation of Dijkstra's algorithm, for the way that we have defined the costs, is that the algorithm will find the sequence of events that minimizes the total distance traveled (by both robots). This could be a useful metric if we were, for example, trying to minimize energy consumption or the exposure of the robots. However, we actually wish to minimize the time it takes to complete the mission. The base algorithm doesn't accomplish this because it doesn't recognize that the events are performed by two different robots that can act in parallel. For example, the application of the standard Dijkstra's algorithm to the given scenario might produce a sequence of events that commands robot A to perform all four tasks, when in reality, the total execution time could be reduced by commanding robot A and robot B to work in parallel. In order to achieve this behavior, the algorithm was modified as follows. Within the algorithm, instead of keeping a running total of the minimum total cost it takes to reach each state, the running cost of events associated with each of the two robots were kept separately. Then the "optimal" path was chosen to minimize the maximum of the two running totals. This change accounts for the fact that the two robots can operate at the same time.

Since so many of the costs assigned to events in our graph are zero, there will likely be many paths through the automaton that possess the minimum cost. For example, there may be multiple strings of events with the same ordering of start events, but with different interleavings of entry and finish events. In order to address this, when presented with a choice between performing a start event (with non-zero cost) and waiting for an entry or finish event (with zero cost) to occur, we always perform the start event. Furthermore, the robot control software is structured to call the planning algorithm to perform an optimization any time a choice between enacting different start events needs to occur. Performing the algorithm at these points improves performance in that it re-optimizes the plan based on the current robot locations, rather than the prediction made based on straight-line distances.

VI. Simulation Results

The Player/Stage open-source software [17] was used for simulation of the test case. This setup consists of two pieces of software working together. The Player part is a defined set of interfaces and drivers that can run in combination with Stage, or an actual robot. The Stage simulation part receives the commands from Player and responds as the actual device would.

The scenario described in Section III was employed for the simulation, where Fig. 1 provides an illustration of the scenario. The field itself is defined to have a size of 40×40, where the intersection of the four regions is defined as the origin of the rectangular coordinate system. Simulation results will be provided for the specific task and initial robot locations defined in Table I.

The optimization algorithm as defined in Section V chose robot A to perform tasks 3 and 4 and robot B to perform tasks 1 and 2. The entry events are controllable, but were not chosen by the high-level control algorithm, rather, the low-level D* Lite algorithm was allowed to choose the path to each task location and the entry events events were just

TABLE I
Locations corresponding to provided simulation results

Location	x-coordinate	y-coordinate
robot A start	-9	-19
robot B start	9	-19
task 1	-2	-5
task 2	7	8
task 3	-5	3
task 4	-13	8

allowed to happen (as if uncontrolled). The problem with this is that the low-level D* Lite algorithm has no knowledge (as it stands) of the high-level and may choose a path that violates the given specifications. For this example simulation, no specifications were violated, but in the future, the low-level algorithms need to be modified to force each robot to stop at a region boundary if the region to be entered is occupied. Additionally, the low-level algorithms need to be modified to choose paths that adhere to the region entry events allowed by the high-level supervisory controller.

For the specific scenario described, the robot system followed the sequence of events $a3s \rightarrow b1s \rightarrow a6e \rightarrow b5e \rightarrow a3f \rightarrow a4s \rightarrow b1f \rightarrow b2s \rightarrow b8e \rightarrow a4f \rightarrow b7e \rightarrow b2f$. An illustration of the animation generated by the Stage simulation environment is shown in Fig. 7.

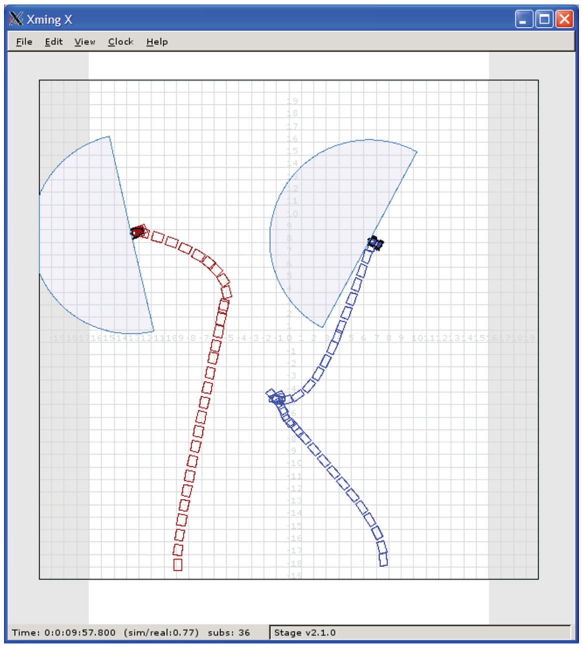


Fig. 7. Animation for the simulation of the given scenario

VII. Conclusion and Future Work

This paper describes the development of a process and tools for synthesizing real-time control software for multiple robot systems that is able guarantee the achievement of a set of formal specifications by construction. The process was then applied to a simple, but illustrative scenario involving the cooperative control of two robots. The process and lessons described in this paper lay the foundation for the application of these techniques to more complex scenarios from a range of fields. In addition to attempting other applications, other future work includes improving the optimality of the resulting behavior and reducing the computational complexity associated with the approach.

Further Applications: In terms of robot applications, one natural direction is to apply this approach to a similar scenario, just with more robots, more tasks, and a larger field. To give an example of the kind of growth that may be expected, the supervisory controller generated for an example the same as the one explored in this paper, but with nine different regions, is represented by an automaton with 5749 states and 25453 transitions. Another scenario to be explored is one where one of the robots is classified as an "enemy" and none of its actions can be controlled. Such a scenario would require that the controllable robots be able to abort a given task and be able to move to locations that aren't associated with completing one of the given tasks. Another direction is to apply these techniques to examples from other fields, like manufacturing, that involve uncontrollable events and a need to optimize over the allowed set of behaviors.

Improving Optimality: In the application described in this paper, the costs assigned to the various events did not accurately reflect the actual time it takes for their enactment. For one, the cost for start events were based on straight-line distances to the associated task, rather than on the actual path taken (to avoid obstacles, illegal regions, etc.). Furthermore, start events were always enacted immediately when available, rather than sometimes waiting for an entry or finish event to occur first, in order to enact a different start event. Making the costs more realistic (to make better decisions) would require better coordination between the high-level planning and low-level planning algorithms, Dijkstra's algorithm and D* Lite, respectively.

Reducing Computational Complexity: As mentioned previously, as the complexity of the application increases, the computation required by the various algorithms can quickly become prohibitive. One way to reduce the complexity is to base optimization decisions on a reduced-order model of the supervisor automaton. For example, in this paper, control decisions were based solely on the starting events. Therefore, an automaton with the entry and finish events projected away can be generated. Such an automaton will often have a reduced state size, though in some cases the state size will actually be larger [12]. Another option to reduce computational complexity is to reuse information from previous iterations of the high-level optimization algorithm each time it is invoked. Such an approach is employed by the D* algorithm and the idea has even been applied to task-level planning in [5]. Knowledge from the low-level path planning algorithms could also be used to reduce the number of high-level events that need to be considered in the high-level automata.

In the field of discrete event theory, an increasing body of results also have been generated recently with regard to applying abstraction [18], [19], [20], [21] and imposing additional requirements that allow global goals to be satisfied locally [22], [23], [24]. Such techniques can greatly reduce the size of the controller automata that must be considered.

References

[1] prepared by Army Capabilities Integration Center Tank-Automotive Research and D. E. C. R. Initiative, "Robotics strategy white paper," March 19 2009.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2002.

[3] M. Kloetzer and C. Belta, "Distributed implementations of global temporal logic motion specifications," in *2008 IEEE Int'l Conference on Robotics and Automation (ICRA)*, Pasadena, USA, May 2008.

[4] M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta, "Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees," in *2010 IEEE Int'l Conference on Robotics and Automation (ICRA)*, Pasadena, CA, May 2008.

[5] S. C. Livingston, R. M. Murray, and J. W. Burdick, "Backtracking temporal logic synthesis for uncertain environments," in *2012 IEEE Int'l Conference on Robotics and Automation (ICRA)*, St. Paul, USA, May 2012.

[6] P. Roy, P. Tabuada, and R. Majumdar, "Safety-guarantee controller synthesis for cyber-physical systems," *CoRR, abs/1010.5665*, 2010.

[7] W. Thomas, *Automata on infinite objects*, ser. Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics. Elsevier, The MIT Press, 1990, pp. 134–191.

[8] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, January 1989.

[9] J. Thistle and W. Wonham, "Control of infinite behavior of finite automata," *SIAM Journal of Control and Optimization*, vol. 32, no. 4, pp. 1075–1097, July 1994.

[10] E. Roszkowska, "Decentralized motion-coordination policy for cooperative mobile robots," in *Proc. of the 2008 Int'l Workshop on Discrete Event Systems - WODES'08*, Gothenburg, Sweden, 2008, pp. 364–369.

[11] K. T. Seow, M. T. Pham, C. Ma, and M. Yokoo, "Coordination planning: Applying control synthesis methods for a class of distributed agents," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 2, pp. 405–415, March 2009.

[12] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems – Second Edition*. Springer, 2007.

[13] UMDES and DESUMA, "Software tools for discrete event systems," `http://www.eecs.umich.edu/umdes/toolboxes`.

[14] S. Koenig and M. Likhachev, "D* Lite," in *18th National Conference on Artificial Intelligence*, St. Paul, USA, May 2002.

[15] I. Ulrich and J. Borenstein, "VFH*: Local obstacle avoidance with look-ahead verification," in *IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, USA, 2000.

[16] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, Inc., 1998.

[17] Player/Stage, "Free software tools for robot and sensor applications," `http://playerstage.sourceforge.net`.

[18] H. Flordal and R. Malik, "Modular nonblocking verification using conflict equivalence," in *Proc. of the 2006 Int'l Workshop on Discrete Event Systems - WODES'06*, Ann Arbor, USA, 2006, pp. 100–106.

[19] L. Feng and W. Wonham, "Supervisory control architecture for discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 53, no. 6, pp. 1449–1461, 2008.

[20] R. C. Hill and D. M. Tilbury, "Incremental hierarchical construction of modular supervisors for discrete-event systems," *Int. J. Control*, vol. 81, no. 9, pp. 1364–1381, September 2008.

[21] R. C. Hill, D. M. Tilbury, and S. Lafortune, "Modular supervisory control with equivalence-based abstraction and state-based conflict resolution," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 20, no. 1, pp. 491–498, 2010.

[22] R. J. Leduc, B. A. Brandin, M. Lawford, and W. M. Wonham, "Hierarchical interface-based supervisory control–part I: Serial case," *IEEE Trans. Automatic Control*, vol. 50, no. 9, pp. 1322–1335, September 2005.

[23] R. J. Leduc, M. Lawford, and W. M. Wonham, "Hierarchical interface-based supervisory control–part II: Parallel case," *IEEE Trans. Automatic Control*, vol. 50, no. 9, pp. 1336–1348, September 2005.

[24] R. C. Hill, J. E. R. Cury, M. H. Queiroz, D. M. Tilbury, and S. Lafortune, "Multiple-level hierarchical interface-based supervisory control," *Automatica*, vol. 46, no. 7, pp. 1152–1164, 2010.