CSL: A Language to Specify and Re-Specify Mobile Sensor Network Behaviors

Karl Hedrick, Jerry Jariyasunant, Christoph Kirsch, Joshua Love, Eloi Pereira, Raja Sengupta, Marco Zennaro
Center for Collaborative Control of Unmanned Vehicles
Email:jlove@me.berkeley.edu, jjariyas@berkeley.edu

***Abstract*—The Collaborative Sensing Language (CSL) is a high-level feedback control language for mobile sensor networks (MSN). It specifies MSN controllers to accomplish network objectives with a dynamically changing ad-hoc resource pool. Furthermore, CSL is designed to allow the updating of controllers during execution (patching). This enables hierarchical control with simpler controllers at lower levels. The CSL Execution Engine contains the intelligence to allocate resources to tasks dynamically and adjust in real time to resource motion, this enables CSL controllers to be simple, intuitive and scalable. Experimental results show that the CSL Execution Engine performs these services with the addition of very little overhead.**

I. Introduction

This paper introduces the Collaborative Sensing Language (CSL), a language for the feedback control of distributed ad-hoc mobile sensor networks (MSN). CSL is comprised of two sub-languages: CSL-MCL the declarative Mission Control Language, and CSL-RPL the imperative Run-time Patching Language. The syntax of the language, its interactions, its motivations, and results from its initial implementation will all be discussed.

CSL's research efforts, and its two sub-languages, are motivated by two coupled problems: the need to program controllers for distributed ad-hoc mobile sensor networks [1]–[8], and the desired ability to significantly change those controllers during execution [9], [10]. Both abilities simplify the control of mobile sensor networks and are therefore necessary parts of CSL.

The first problem arises due to human-user limitations. A single user cannot properly control the low-level details of multiple resources concurrently. Thus, the user should only state network objectives and receive only information relevant to those objectives. This is further complicated if a network may have variable dimension, i.e., sensors and channels come and go. Therefore a goal oriented network controller interacting with a dynamic network should be weakly coupled to the size of the network. A tightly coupled controller, i.e., one reacting at fine grain to changes in network dimension, would be too complex to scale. MCL controllers are very weakly coupled for scalable execution by large networks. MCL controllers do not specify the allocation of tasks to platforms, coordination structures between collaborating platforms, or lower-level platform controllers. All of this is embedded in the CSL Execution Engine. It is part of the network intelligence, similar to the Internet which autonomously adapts to the loss of a router without troubling its applications. This helps reduce the cognitive workload required of the user.

The second problem requires the run-time patching of controllers and has value in many ways. Controllers are typically continuously operating components; think of air traffic management systems, traffic signal control systems, or Google servers. Updates should be run-time updates unless one assumes it is acceptable to leave the plant without control for some time. In the MSN context, the ability to change controllers at run time leads to simpler controllers. Instead of defining a controller able to accomodate all eventualities, one may set up the current controller for a short horizon, and patch it to a new controller if the state changes in unforeseen ways.

One alternative would be to develop a specific compiled controller that is executed, when changes are necessary to it the system must be halted, a new controller compiled, loaded and re-executed as in [3], [6]. This may be viable in several situations, but not in a continuously operating mobile sensor network.

The run-time patching idea fits naturally with hierarchical control. A higher-level controller can change a lower-level

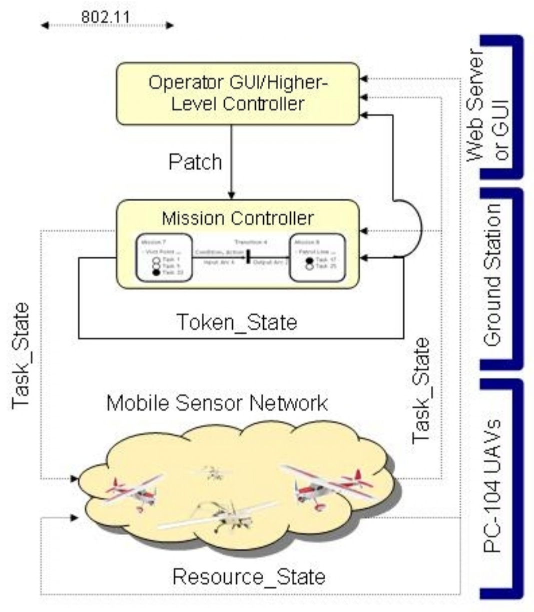


Fig. 1. Feedback Control of Sensor Networks

controller at run time based on its broader understanding of the state and the intended system performance.

MCL is intended to describe a controller for the mobile sensor network. Syntactically MCL is a Petri net style language. A sentence in MCL declares a specific controller for the MSN. The CSL-MCL Execution Engine interprets and executes the MCL to form a MSN controller within the "Publisher"component of the system. The MSN controller goes into feedback coupling with the MSN in the classical way as illustrated by Figure 1. The MSN is viewed as the plant, sending observations (Task_State) to the MSN controller, which in turn sends actuation commands (an updated Task_State) to the plant. The complete CSL state contains the Task_State as well as a Resource_State and a Token_State. For a MCL controller, changes in the Resource_State (sensors moving) cause changes in the Task_State (tasks being executed). The completion of tasks cause changes in the Token_State which in turn causes changes in the Task_State (new tasks to be completed). The new tasks cause changes in the Resource_State (new tasks allocated to resources). This process is illustrated in Figure 1.

MCL is a declarative language targeting a network model of computation. This means two things:

1) Tasks can execute concurrently unless constrained by resource availability or the MCL program.
2) The set of resources is interpreted as a finite but arbitrarily large set of objects. This ad-hoc set changes dynamically during program execution as sensors move within, join or leave the network.

The first point distinguishes MCL from Petri net based languages such as [5], [11]. The second point differentiates MCL from the mission definition language CCL [1], [2] and the MARS2020 demonstration [3] which used predefined groups. It also distinguishes MCL from [4], [12]. The SHIFT simulation language is designed for controllers targeting networks [13]. Its "exists"and"all "quantifiers appear in CSL. CSL is currently less expressive to enable an initial implementation.

The run-time patching language (RPL) is intended to patch the MSN controller dynamically. Thus the MSN controller can be changed on-the-fly as it is operating. This is done by patching an already operating MCL program at run time. An external controller or operator, as shown in Figure 1, can dispatch an RPL program to the CSL Execution Engine to reduce, grow or otherwise alter the MSN controller at run time.

CSL-RPL is an imperative language, and as such, a sentence in RPL is interpreted at run time to act on the current MSN controller and transform it into a new MSN controller. This enables heirarchical control of the MSN. The computation mapping the old MSN controller to the new controller is the semantics of RPL. The on-the-fly re-programming of control differentiates CSL from Mission Lab [6], MRL [7], and others such as [8], [14]–[16].

Section II introduces the declarative CSL-MCL to represent the MSN controller. Section III introduces CSL-RPL which is used to change the controller described in Section II. Section IV describes a first generation implementation of the CSL Execution Engine and its associated simulation environment. Section V provides experimental results which show that the overhead associated with the first generation CSL Execution Engine is insignificant and improving the network performance reduces to attaining improved solutions for the NP-hard open travelling salesman problem. Finally, Section VI provides conclusions and future work.

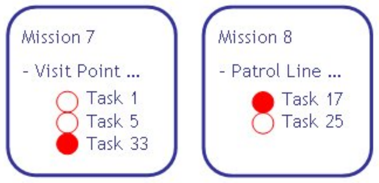


Fig. 2. Example Mission Controller

II. CSL-MCL

CSL-MCL allows users to specify rich, state-dependent, evolving network missions using high-level descriptions. The CSL Execution Engine automatically handles lower-level functions like resource allocation and reliable task completion. CSL-MCL programs can be created and modified graphically (or textually by a user/higher-level controller) and stored in the XML based CSL-MCL syntax that is executed as described below. XML was chosen to improve interoperability between languages and operating systems and also to enable web services for machine-machine and human-machine interactions.

A. *Mission Controller Specification*

MCL programs are composed of missions. Figure 2 shows a program with two missions, named Mission 7 and Mission 8, represented by the two boxes. A mission is an instance of a mission type. Mission 7 is of type "visit point ". The point to be visited is a run-time parameter and is specified when instantiating Mission 7 from the type "visit point". Mission 8 is of type "patrol line".

A mission type denotes a type of sensor behavior with parameter values to be set during the mission's run-time creation. The sensors in the network must know how to execute the CSL mission types, but each may do so in a different manner with a different low-level controller. For example, both a UAV and a UGV can visit a point, but they do so in different fashions. Table I lists the current mission types. They are travel and track for 0, 1, or 2 dimensions. Travel missions finish once all of the points or area has been traversed. Track missions do not end until terminated by an external controller using RPL.

Missions execute by associated tasks. For example, Mission 7 in Figure 2 has three tasks. All the tasks within a given

TABLE I
CURRENT PRIMITIVES

Type	0-D	1-D	2-D
Travel	visit point	visit line	sweep area
Track	loiter point	patrol line	patrol area

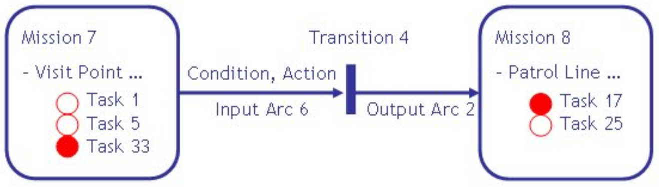


Fig. 3. Mission Controller with Sequencing

mission (box) have identical behaviors. Thus tasks 1, 5, and 33 of Mission 7 represent three different visits to the same point. The three tasks may be executed by one, two, or three different mobile sensors. This allows new copies of the task to be created without ending previous executions or removing their historical execution information. This is useful for missions that are cyclical where a user may not want to terminate the completion of one task to start the execution of an identical copy. The filled/empty nature of the circles next to the tasks will be discussed shortly.

Missions can be specified to execute concurrently or sequentially. Missions 7 and 8 in Figure 2 would execute concurrently. Figure 3 shows how to force sequential execution. This is done, in a Petri net style, by connecting both missions to a transition. As in Petri nets, transitions are the gates that control the flow of tasks. The direction of the flow is controlled through the placement of input and output arcs. The non-Petri net aspects are the condition on the input arc and certain aspects of the actions discussed shortly.

Input arcs are directed links from missions to transitions. Input arcs are associated with a condition/action pair. Conditions are guards on the input arcs. In order for a transition to fire, which will be discussed shortly, all conditions on all input arcs to that transition must be satisfied. This is analogous to a compound *if-then* statement: *if* α AND *if* β AND *if* γ ... *then* FIRE. The satisfaction of each input arc is part of the compound if clause and is based on the current state of the connected mission. Once the conjunction of all input arcs is satisfied, the transition can be fired, and all of the associated actions can be taken on the connected input missions.

Conditions return booleans that are either true or false signifying if the condition is met or not. The first generation implementation has pre-programmed conditions based on the state of tasks (To Do, Assigned, Done, Cancelled) and global timers. For example, one can program a condition such as "when any one of the tasks in Mission 7 is done". In future implementations of CSL this will be expanded so that any external function may also produce a boolean for a condition expression based on the CSL state. This would allow for more complex expressions such as "if you see a white Chevy Tahoe in a picture captured ", which requires image processing external to CSL.

These pre-programmed conditions are evaluated over the state of tasks with tokens in the mission box. A filled circle in a mission box next to a task number indicates a token. A token is always associated to a task but not all tasks are associated to tokens. For example, in Figure 3, only tasks 17 and 33 have tokens. The lack of a token (empty white circle) in Task 1 and Task 5 indicates that the tokens in these tasks have been removed by past firings of Transition 4. The lack of a token in Task 25 could be due to an RPL command (such as cancel). The token is used as a flag to signal if a task has or has not already been used to fire a transition. This is needed since once certain conditions are satisfied by a given task, they will always be true. For example, "if Task 1 is done"is always true once Task 1 is completed. The presence of the token records that the task has not yet been used to fire a transition and may do so in the future. The list of all tokens is the third state item, the Token_State.

Actions on an input arc specify the removal of tokens. Note that the removal of tokens has no impact on the state of any task. Tokens merely regulate the temporal creation of tasks. Thus in Figure 2, tasks 1, 5, and 33 may all be executing even though only Task 33 has a token. The other tokens may have been removed by the user with RPL.

An action on an input arc can remove one token or all tokens when a transition fires. In traditional Petri nets, a single token is removed for each unit-weight input arc when a transition fires, other fixed arc weightings are possible to remove multiple tokens. A CSL input arc with the action remove(all) can remove an arbitrary number of tokens, namely however many exist in that mission, which may not be known a priori.

Input arcs remove tokens while output arcs add them. When a transition "fires"each output arc creates a token and task in the mission to which it is going. This is like a unit weight Petri net.

The syntax for representing CSL-MCL Automations is formally defined in an XML format. The following example is a mission with id=7, that is defined to be a visit point. The mission is then followed by a token/task.

```
<MISSION mid="7">
    <TRAVEL DIMENSION="0" REFERENCE="INERTIAL">
        <POINT LAT="35.7163" LON="-120.7679" />
    </TRAVEL>
</MISSION>
<TOKEN mid="7" id="352" />
```

This XML format must conform to the CSL-MCL document type definition (DTD), the final part of which is below.

```
<!ELEMENT MISSION (( TRANSITION* , IARC* , OARC* ,
MISSION* ) |TRAVEL |TRACK)>
<!ATTLIST MISSION
mid ID ♯ REQUIRED>

<!ELEMENT TRAVEL (POINT+)>
<!ATTLIST TRAVEL
dimension (0 |1 |2) ♯ REQUIRED
reference CDATA ♯ REQUIRED
priority CDATA ♯ IMPLIED
number_agents CDATA ♯ IMPLIED>

<!ELEMENT TRACK (POINT+)>
<!ATTLIST TRACK
dimension (0 |1 |2) ♯ REQUIRED
reference CDATA ♯ REQUIRED
```

```
priority CDATA ♯ IMPLIED
number_agents CDATA ♯ IMPLIED>

<!ELEMENT POINT EMPTY>
<!ATTLIST POINT
lat CDATA ♯ REQUIRED
lon CDATA ♯ REQUIRED
alt CDATA ♯ IMPLIED>

<!ELEMENT TRANSITION EMPTY>
<!ATTLIST TRANSITION
tid ID ♯ REQUIRED>

<!ELEMENT IARC (CONDITION, ACTION)>
<!ATTLIST IARC
aid ID ♯ REQUIRED
from REF ♯ REQUIRED
to REF ♯ REQUIRED>

<!ELEMENT CONDITION EMPTY>
<!ATTLIST CONDITION
type (onSTATUS, onTIMER) ♯ REQUIRED
value (todo |assigned |done|CDATA) ♯ REQUIRED
scope (task |mission) ♯ REQUIRED>

<!ELEMENT ACTION EMPTY>
<!ATTLIST CONDITION
type (none, remove, removeAll) ♯ REQUIRED>

<!ELEMENT OARC EMPTY>
<!ATTLIST OARC
aid ID ♯ REQUIRED
from REF ♯ REQUIRED
to REF ♯ REQUIRED >

<!ELEMENT TOKEN EMPTY>
<!ATTLIST TOKEN
mid ID ♯ REQUIRED
id ID ♯ REQUIRED
tid IDREF ♯ IMPLIED>
```

The document type definition (DTD) contains elements for missions, transitions, input arcs, conditions, actions, output arcs, and tokens. The DTDs will be changed to .xsd schema files for a second generation implementation.

B. State Representation

During the execution of a CSL-MCL program there must be some concept of state. This CSL-MCL state contains three items: the state of the automation (Token_State), the state of the tasks (Task_State) and the state of the sensor resources (Resource_State).

The state of all tasks in the system, known as the Task_State, has evolved from [17], [18]. The Task_State contains the definitions and execution information for the tasks that are created by the MCL controller then assigned and executed by the mobile sensor network. It enables the user to understand the recent and expected network progress. CSL's Task_State structure is defined below as an XML DTD.

```
<!ELEMENT Task_State (TASK*)>
<!ATTLIST Task_State
agent_ID CDATA ♯ REQUIRED
send_time CDATA ♯ REQUIRED>

<!ELEMENT TASK EMPTY>
<!ATTLIST TASK
task_id ID ♯ REQUIRED
mid IDREF ♯ REQUIRED
tstatus (TODO |ASSIGNED |CANCELLED |DONE) ♯ REQUIRED
status_time CDATA ♯ REQUIRED
create_time CDATA ♯ REQUIRED
agent_ID CDATA ♯ IMPLIED
step CDATA ♯ IMPLIED
cost CDATA ♯ IMPLIED>
```

The first element represents the Task_State. Each Task_State message contains the set of tasks and an agent_ID and send_time to determine by whom and when the message was sent. The second element represents the syntax for an individual task. Each task has a required unique identifier and is associated with a single mission. Each task also has a creation time. During execution the previously mentioned fields stay fixed; however, the status, agent_ID, step, and cost all change. When this change last occured is recorded in the status_time. The agent_ID is the identifier of the agent assigned to this task. The step is which step the task is in the agent's multi-step plan, and the cost is the cost (based on distance) to complete that individual step. The Task_State contains all of the required information to describe the state of the current list of tasks active in the mobile sensor network.

The second state item described is referred to as the Resource_State. This includes: agent locations, task assignments and vehicle status (fuel, battery, errors, etc.). It is important for a complete understanding of the physical system state. For example, an CSL-MCL controller should not ask for 30 geographically sparse locations to be visited with a single UAV, but if there are several UAVs, this may be a reasonable request. The resources available may also change during execution since UAVs can join, leave, and move. All of this feedback information is available in the CSL format for display. The Resource_State is syntactically defined below and is part of the CSL-MCL DTD.

```
<!ELEMENT Resource_State (RESOURCE*)>
<!ATTLIST Resource_State
agent_ID CDATA ♯ REQUIRED
send_time CDATA ♯ REQUIRED>

<!ELEMENT RESOURCE EMPTY>
<!ATTLIST RESOURCE
agent_id ID ♯ REQUIRED
task_id IDREF ♯ IMPLIED
status (idle |assigned) ♯ REQUIRED
status_time CDATA ♯ REQUIRED
lat CDATA ♯ REQUIRED
lon CDATA ♯ REQUIRED
alt CDATA ♯ REQUIRED
pitch CDATA ♯ IMPLIED
roll CDATA ♯ IMPLIED
yaw CDATA ♯ IMPLIED
velocity CDATA ♯ IMPLIED
fuel_level CDATA ♯ IMPLIED
battery_level CDATA ♯ IMPLIED
error_message CDATA ♯ IMPLIED>
```

Again, the structure breaks down into two different elements. The first element, Resource_State, is the overall list of resources (in the current implementation UAVs but in the future hopefully UGVs, AUVs and stationary sensors) and the associated information of who this message is from (agent_ID) and when it was sent (send_time). The second element is the actual structure for individual resources. Obviously information like the agent's ID, status, and position are required. Here the status_time indicates the last time any of the fields were updated, not just changes in the assigned/idle nature of the resource. Other information may also be available and transmitted, such as the task assigned (if one is), orientation, and other resource information like fuel and battery levels.

The third and final state item is the token distribution. The current CSL-MCL implementation utilizes Petri nets as a representation model for this automation, so the Token_State is the distribution of tokens in the Petri net. Other alternative models for representing the automation could be utilized, but the Petri net adequately represented the concurrent nature of task creation.

C. Mission Evolution

Mission evolution will be illustrated with several examples. Figure 4 defines a mission starting with a visit line (e.g. a border) followed by visits to two separate points.

Assume the operator wants two UAVs to visit the border, this is specified by placing two tasks with tokens in Mission 1. This is done using an RPL command explained in the next section. If there are at least two idle sensors in the network the CSL Execution Engine will ensure the two tasks are picked up and executed concurrently by a pair of sensors. Since the condition on the arc explicitly states that it is the mission that must be done (i.e. onStatus = done with scope=mission instead of scope=task) once the states of both tasks 1 and 2 are done, the transition is fired.

The state of a task is controlled by the MSN. It is uncontrollable but observable by the CSL-MCL controller. The user does not state when a task is assigned or who it is assigned to, but knows this information once available.

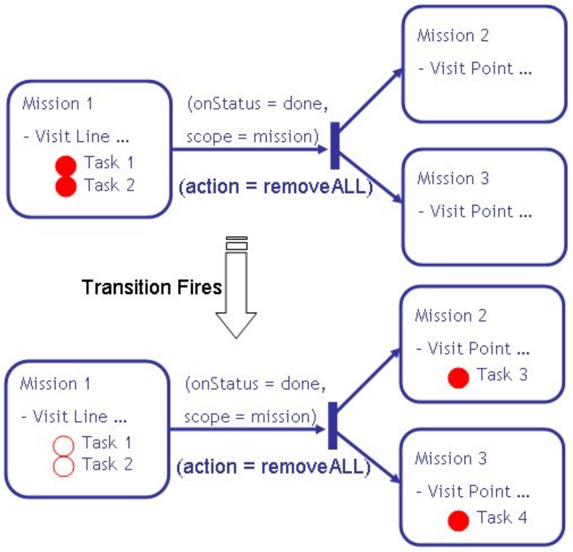


Fig. 4. Mission Controller with Constrained Concurrency

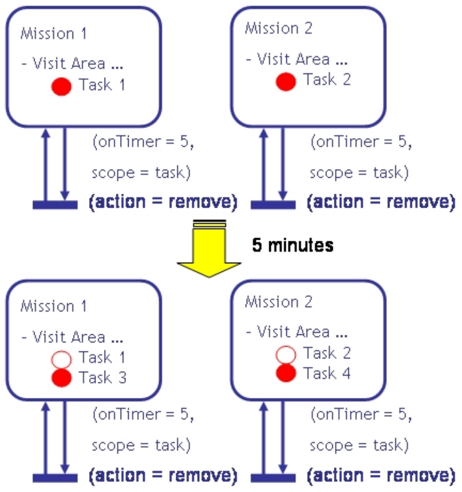


Fig. 5. Modified Repeated Script

In Figure 4, when tasks 1 and 2 are done, the corresponding input arc action removes both tokens in Mission 1. The two output arcs add one new token and task to Missions 2 and 3 respectively. The two new tasks would then be automatically allocated to network sensors by the CSL Execution Engine. Note that the CSL input arc action remove(all) cannot be mapped to a Petri Net arc with non-unit weight since the number of tokens to be removed is not known a priori.

For a second example, refer to Figure 5. Assume only one UAV is available and two areas must be visited periodically. At the beginning (top) of execution of Figure 5 there is only Task 1 and Task 2. Since there is only one UAV, it must visit Area 1 and then Area 2 (or in the other order). After 5 minutes, tasks 3 and 4 are created and again they are visited one after the other. The order in which they are visited is determined by the task allocation mechanism, Section IV.B. If instead there are two or more available UAVs the tasks would be executed in parallel. While the intent is for the tasks to be completed in parallel, if the system is limited by its resources it does the best it can to accomplish the objectives as quickly as possible.

The mission evolution can also be described as traces of the state. The CSL-MCL state can be represented by the triple (r,t,a) where "r"is the state of the resources, "t"is the state of the tasks, and "a"is the state of the automation (distribution of tokens). The following illustrates the evolution of Figure 5 for the single UAV case.

$$\begin{matrix} r_0 \\ t_0 \\ a_0 \end{matrix} \xrightarrow{assign} \begin{matrix} r_1 \\ t_1 \\ a_0 \end{matrix} \xrightarrow{move} \begin{matrix} r_2 \\ t_1 \\ a_0 \end{matrix} \xrightarrow{complete} \begin{matrix} r_2 \\ t_2 \\ a_0 \end{matrix} \xrightarrow{move}$$

$$\begin{matrix} r_3 \\ t_2 \\ a_0 \end{matrix} \xrightarrow{complete} \begin{matrix} r_3 \\ t_3 \\ a_0 \end{matrix} \xrightarrow{Loiter} \begin{matrix} r_4 \\ t_3 \\ a_0 \end{matrix} \xrightarrow{fire} \begin{matrix} r_4 \\ t_4 \\ a_1 \end{matrix} \xrightarrow{repeat} \ldots$$

The system begins at some intial state with some initial position, set of tokens and set of tasks. The resouce becomes assigned to the tasks changing r_0 to r_1 and t_0 to t_1, the assignment is recorded in both locations for bi-directionality. The UAV then moves to complete one of the tasks, this also changes the Resource_State. Once the task is completed the Task_State changes and the resource begins to move again to the next task. The UAV completes the second task and then begins to loiter. Finally, the 5 minute cycle time is reached and the Publisher fires the transition to create a new set of tasks, which will repeat the same process.

It is obvious that the resources move continuously and so the Resource_State is continually changing. Also the task and automation states change as discrete events. The state evolution assumes a zero-time computation for token and task changes, i.e. the creation of a new task is instantaneous.

III. CSL-RPL

The Run-time Patching Language enables operators to modify the mission controller dynamically by editing the corresponding MCL at run time. This is illustrated using the example in Figure 6.

At the beginning of run shown in Figure 6, t_0 to the left, there is no controller and no tasks are being executed. At time t_1 the user/high-level controller decides to request that an area be patrolled. MSN state feedback and video streams are provided to the user. Eventually, at time t_2, the user identifies an object of interest at a location. Then at t_3 the user can use RPL to terminate the area patrol and add a visit point to look at the location of interest. At t_5 the visit location is updated, possibly to modify its GPS coordinates because the object has moved. Finally, more information continues to stream in even after the task is finished at t_6. For a more specific example imagine searching an area for a specific truck and a suspected convict, once the truck is seen in the area, revisit the truck. Now consider that the user may not have specified the correct point, mistakes do happen, so simply correct the task to what was intended. The ability to patch the mission controller during execution enables heirarchical control, correcting for mistakes made and adjusting for unpredictable situations. This helps to reduce the need for complex controllers that anticipate every feasible outcome.

RPL is an imperative language with three commands: add, delete, and update. An RPL program is a sequence of such commands. Using these commands an external controller or operator can change the current mission controller at run time, while the controller is running in feedback with the sensor network. More specifically, the imperatives can add, delete, or modify the syntactic elements in MCL, i.e., mission definitions, transitions, input arcs, and output arcs. This is very similar to [9] which uses patching to modify lower-level real-time controllers.

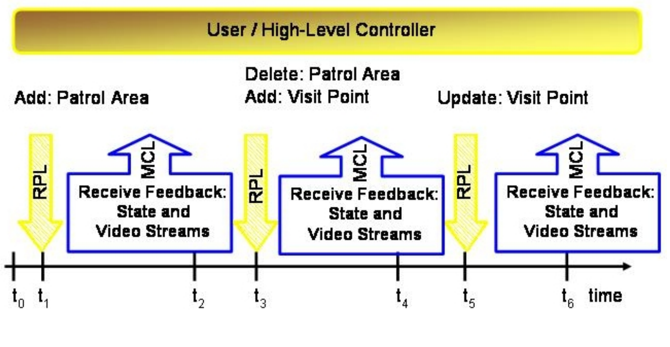

User / High-Level Controller
Add: Patrol Area
Delete: Patrol Area
Add: Visit Point
Update: Visit Point
RPL
MCL
Receive Feedback: State and Video Streams
RPL
MCL
Receive Feedback: State and Video Streams
RPL
MCL
Receive Feedback: State and Video Streams
t0
t1
t2
t3
t4
t5
t6
time

Fig. 6. Run-Time Patching Example

Setting up the first mission controller and endowing it with the initial tokens is also done via RPL. The Null controller is the valid starting controller and all controllers are derived from it by a combination of RPL programs.

A run-time mission update is not the same as a delete followed by an add. When a mission is deleted, all tasks associated with it are cancelled. The sensors executing those tasks are released back to the allocation system which may allocate them to other tasks. Adding the mission back will then re-engage the allocation system in once again allocating a possibly new set of sensors to the mission. By contrast, an update could modify task parameters without affecting the allocation of sensors to the mission being updated.

The updating option is available only for missions and incoming arcs since there is no information carried on transitions and outgoing arcs that would make updating them necessary. A transition is defined only by a unique identifier, which has no information to be updated. An outgoing arc is defined by an identifier, the transition that it is from and the mission that it is to. Updating any outgoing arc is the same as deleting and then adding a new outgoing arc, so in this situation updating is equivalent to a delete followed by an add. With updates users can modify mission parameters that should be stored (e.g., the GPS locations defining an area for a mission of type search area), arc actions and arc conditions during the execution of the mission controller.

Again, RPL's syntax is defined as an XML DTD:

```
<!ELEMENT ADD_ACTION (MISSION |TRANSITION |IARC |OARC
|TOKEN |TASK)>

<!ELEMENT DELETE_ACTION EMPTY>
<!ATTLIST DELETE_ACTION
id ID ♯ REQUIRED>

<!ELEMENT UPDATE_ACTION (MISSION |IARC)>
<!ATTLIST UPDATE_ACTION
id ID ♯ REQUIRED>
```

This obviously contains the three elements of add, delete, and update. For each addition, the MCL structure to be added is passed. For each deletion, the identifier of the element to be deleted is passed. For each update, the new updated MCL structure to replace/update an existing structure is passed.

IV. CSL Execution Engine

The CSL Execution Engine accepts RPL files, interprets them into MCL based controllers and executes these controllers. This Execution Engine contains the Publisher which observes the state information, implements the MCL controller and updates the state, see Figure 1. The Publisher outputs updated state information, including a list of tasks in the Task_State, that is the input to the previously existing Collaborative Control System (CCS) which performs the task allocation [17].

A. Publisher

The CSL Publisher is provided all of the information from the Collaborative Control System about the tasks. The Publisher monitors this information along with RPL changes to appropriately add, update, and delete tasks in the Collaborative Control System.

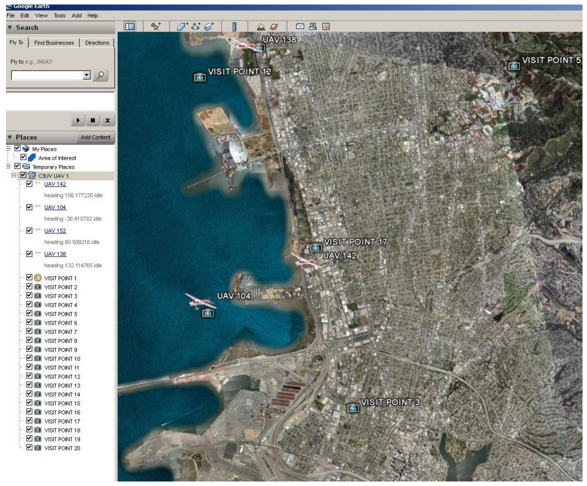
The Publisher has a well-defined operational loop. The loop begins with checking for RPL commands changing the mission controller. Secondly, it checks to see if any transition can be fired. If one can be fired, it is, and old tasks/tokens are removed and new tasks/tokens are added. Finally, the Publisher then checks for updates of the CSL state (Task_State and Resource_State).

At the beginning of the loop, the Publisher checks for any incremental changes to the current control program. Incremental changes are made by adding a new RPL file to a RPL folder. These files start at 1.xml and proceed upwards to 2.xml, etc. These files can be generated by any source and are simply required to be in the appropriate CSL syntax, as validated by the appropriate document type definition or schema file.

Currently the incremental change files can be made manually using WordPad, over the internet using the CSL Web Server (using XML based web services), or with the CSL GUI. The Publisher checks if the next expected file exists. If so, it is parsed with a standard XML parser and the MCL based MSN controller is updated. Since the controller is defined similarly to Petri nets, the current controller definition is stored as lists of missions, transitions, input arcs, output arcs and tokens. If the next incremental file does not yet exist, the Publisher moves to checking for a “fireable”transition. Also, only one incremental update is read per cycle to prevent multiple files from occupying the entire cycle’s duration.

Once any RPL updates have been read, the Publisher runs through the transition list and checks if there is a transition where all incoming arc’s associated conditions are satisfied. If not, it moves on to checking for status updates. If there is one such transition, the transition is fired and tokens/tasks are appropriately added and removed. Also, at most 1 transition may be fired per cycle.

Once RPL changes and transition firings have occurred, the Publisher remains in a communications mode. Here it transmits to the CCS and listens for responses from the CCS. The entire cycle is repeated once .25 seconds have elapsed. The 4 Hz frequency was chosen to coincide with the GPS refresh rate on the UAVs. This provides that changes in scripts and transitions being fired must occur at least .25 seconds apart. This rate could be increased, but mission updates are created by the user and will very rarely come quicker than once a second. Even if updates do come in rapid bursts, the folder where RPL update files are submitted acts as a buffer



My Places
Area of Interest
Temporary Places
C3UV UAV 1
UAV 142
heading 156.177220 idle
UAV 104
heading -36.410782 idle
UAV 152
heading 80.509216 idle
UAV 138
heading 132.114765 idle
VISIT POINT 1
VISIT POINT 2
VISIT POINT 3
VISIT POINT 4
VISIT POINT 5
VISIT POINT 6
VISIT POINT 7
VISIT POINT 8
VISIT POINT 9
VISIT POINT 10
VISIT POINT 11
VISIT POINT 12
VISIT POINT 13
VISIT POINT 14
VISIT POINT 15
VISIT POINT 16
VISIT POINT 17
VISIT POINT 18
VISIT POINT 19
VISIT POINT 20

Fig. 7. Mission status visualized using Google Earth

and stores these files until they can be read. Similarly, the creation of tasks will rarely occur on the order of a second.

B. Collaborative Control System (CCS)

The Collaborative Control System accepts a list of tasks (Task_State) and allocates individual tasks to UAVs with a multi-step greedy algorithm based on Dubins distances. It provides an efficient allocation mechanism for assigning tasks to UAVs. The costs are continuously recomputed and redistributed among the network. This helps adapt to changes in network topology. When UAVs enter, they begin to bid on tasks. When UAVs exit, they stop bidding on tasks, and their tasks are reallocated. If a UAV suddenly stops responding, a timeout based on cost of the task being previously performed is enforced.

Once individual UAVs have tasks assigned to them, they utilize their on-board low-level controllers to physically execute the tasks. These low-level controllers could be different on each vehicle, but they will attempt to complete the same primitive behavior types, i.e. visit point. While executing, the UAVs disseminate the CSL state information (Task_State and Resource_State) as feedback to the CSL Execution Engine (Publisher) and as feedback to the user to be displayed in whatever manner. The current implemenation displays this information in a Google Earth based display, shown in Figure 7 [18].

C. UC Berkeley UAV Fleet

The University of California Berkeley’s Center for Collaborative Control of Unmanned Vehicles (C3UV) has a fleet that currently includes five Sig Rascal 110s, Figure 8, and five Bat IVs from MLB Technology. The Rascals are a smaller (6 ft wingspan) hour-flight platform used for much of the testing and demonstrations. The Bat IVs are a larger (12 ft wingspan) 8 hour platform with onboard generators to provide power for a payload that can contain up to 25 pounds.

Each of these platforms has been developed to interact with a PC-104 computer stack, using the QNX 6.3 operating

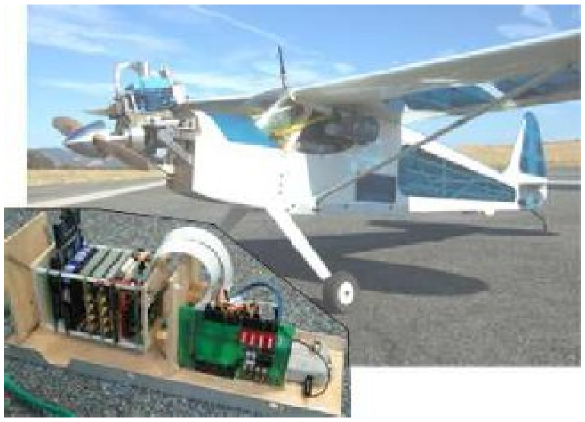


Fig. 8. UC Berkeley Sig Racal and PC-104 Payload

Fig. 9. Simulated Belly-Mounted Camera Feed

system, and a Piccolo autopilot from Cloudcap technologies. The Piccolo autopilot performs all of the low-level actuation and provides a waypoint and turn-rate interface to the PC-104 stack. The Piccolo communicates its information (GPS, orientation, servo values) down to a ground station through a 900 MHz channel, for monitoring during experiments.

The PC-104 computer stacks, Figure 8, contain the onboard CCS control code to determine task allocations. They also have mid-level controllers that accept tasks and produce turn-rate commands for the Piccolo autopilots. These are the implemenations of the controllers that execute each type of task listed in Table I. These values are passed to the Piccolo over a serial connection.

In order to communicate within the mobile sensor network the PC-104s are on an ad-hoc 802.11b network, operating over 2.4 GHz. The Task_State and Resource_State utilize this connection to spread their information throughout the MSN and to the groundstation. The PC-104 also connects to several peripherals to collect image, video, or sensor streams.

The groundstation computer connects to the MSN over the 2.4 GHz ad-hoc network to learn the Task_State and Resource_State. The "Publisher"component resides on this machine and compares the Task_State to its MSN controller to determine if the Task_State should be updated. This computer also typically runs either a CSL GUI or a CSL Web Server, although these could be run on other locally networked computers as well. If an CSL Web Server is used it is typically connected to the external internet for access via any internet device.

D. Hardware-In-Loop Simulations

Since acquiring the airspace to fly experiments is not always easily attainable or necessary, having a high-fidelity simulation environment is quite desirable. The C3UV Hardware-In-Loop simulation environment (HIL) satisfies this need. A 6 degree-of-freedom simulator (written by Cloudcap) provides "fake"position and orientation data to the Piccolo autopilot. In essence, the autopilot thinks it is flying and accepts and issues all commands just as it would during a "real"experiment. These commands are monitored by the simulator which responds as the dynamics would dictate. The only component missing is the physical UAV itself.

All of the computing hardware and code is used, in real time, to respond to the simulation. It provides a very realistic environment. A Google Earth based "belly-mounted camera view"has been written to fake the video stream seen by the UAV, see Figure 9. In totality, all of the code and hardware is run as it would during a real experiment and all of the feedback and information that would be available is still available.

E. Demonstrations

The Generation I CSL Execution Engine has been thoroughly tested in Hardware-In-Loop simulations. It was also demonstrated in the Fall 2007, Spring 2008 Tactical Network Topology (TNT) Program at Camp Roberts, CA, and Fall 2008 Large Tactical Sensor Networks (LTSN) Demonstration in Quantico, VA. Figure 7 shows the CSL integrated Google Earth Feedback display that was used in conjunction with Firefox, Internet Explorer, and Safari on an I-Phone to task the CSL Network via the CSL Web Server interface. Figure 10 shows a photograph collected of the Camp Roberts, CA runway during the Spring 2008 TNT demonstration.

Fig. 10. Collected Picture of Runway at Camp Roberts, CA

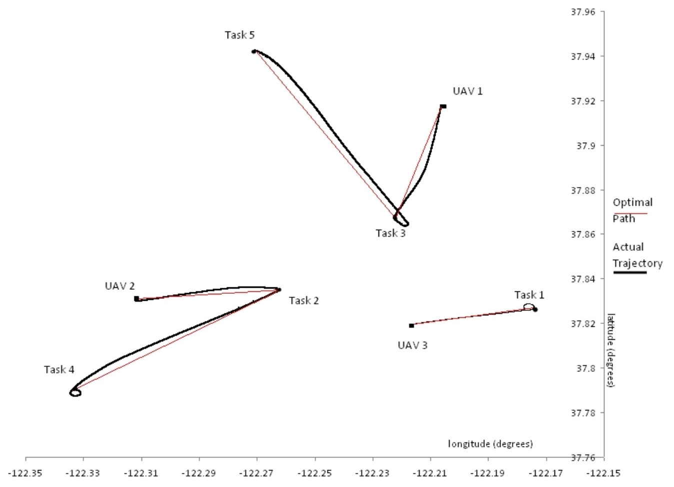


Fig. 11. Example Batch Trajectories

V. Experimental Results

To test the performance of the Generation I CSL Execution Engine, a Hardware-In-Loop simulation was run with the purpose of measuring the overhead time that CSL adds to the system.

The simulation was run with three UAVs executing visit point missions. The starting positions of the UAVs, as well as the locations of the visit points, were randomly generated with a uniform distribution over a 10km by 10km area. Missions were created in batches: over the course of one minute, the missions were generated following a Poisson process with a probability of .1 that a task will be created at any given second. After one minute, all the missions were allowed to be executed until completion before a new batch was sent into the system. The simulation was run for 32 batches; the range for the number of points generated in the simulation was between 2 and 10 tasks for each of the batches.

An example of a batch's execution is shown in Figure 11. The UAVs start at initial positions and fly Dubins paths through the points to be visited. This is compared to the point-mass modelled optimal solution that has no turn dynamics.

Figure 12 shows the non-dimensionalized break-down of total running time of an average mission, which includes three

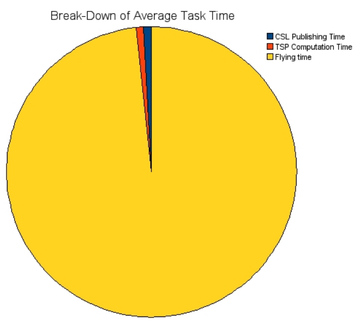


Fig. 12. Distribution of Total Execution Time

TABLE II
CSL Performance

Process	Percent of Time	Mean (sec)	Std. Dev. (sec)
CSL	0.92	3.59	.92
TSP	0.77	3.01	2.46
Flying	98.31	384.74	223.03

processes.

1) The Webserver processing the http request and publishing tasks to the planes
2) The planes determining amongst themselves which one is assigned to the mission
3) The plane assigned to the mission flying to the location and completing the task

The time to execute Process 1 can be considered the overhead added by CSL's infrastructure to the running time of the overall system. Process 2 is dependent on the computation time required to perform an allocation for the open multi-agent traveling salesman problem. The computation of the open multi-agent traveling salesman is a NP-hard problem, and more importantly, it is not solvable in a reasonable time for the planes to effectively accomplish their missions. Therefore, the implementation employs a heuristic algorithm which minimizes the total time necessary to complete all missions [17], [18]. The time to execute Process 3 is dependent on the trajectory taken to reach the destination, subject to a given flying velocity, turning radius, and environmental factors such as wind speed and direction.

As seen in Table II the additional overhead associated with CSL and the heuristic traveling salesman allocation is negligible in comparison to the time required to physically execute the tasks. This is to be expected, the time required to send a web service request over the internet, and have it "published"should be minimal in comparison to the time required to physically perform a task. Also, the standard deviations provide significant insight. The standard deviations for creation and assignment of tasks were small, again as expected since the time required to send a message via the internet should not vary drastically. The standard deviation of time required to complete a task once it is assigned is quite large, but this is merely due to the random nature and distribution of tasks.

Since over 98 percent of the execution time comes as a result of physical motion, the performance of the system is highly dependent on the success of its heuristic allocation mechanism. The details of this multi-step greedy approach are available in [17]. As this allocation approaches optimal, the overall system performance approaches optimal. Reducing the amount of time required for Process 1 and Process 2 overhead would improve the performance, but only slightly. Developing more tightly bounded sub-optimal solutions to the open traveling salesman allocation problem is a continuing field of research.

The Generation I CSL Execution Engine allocates tasks to resources to minimize the maximum time required to complete all tasks [17]. To justify this selection Figure 13 shows a

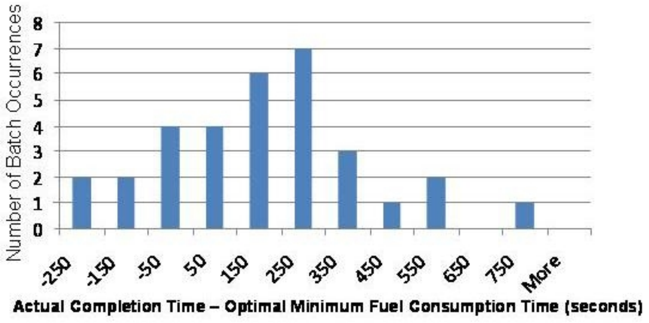


Fig. 13. Comparison of Different Optimizations

comparison between the current allocation mechanism and one that operates a posteriori to minimize the total distance travelled. It is important to emphasize that these are different optimization problems and that one runs during execution while the other takes significant offline computation. The horizontal axis of Figure 13 shows the time difference between the actual performance of the system minimizing the overall time and the optimal performance of an approximated point-mass system if it minimized total distance travelled.

Obviously the actual performance is degraded by the fact that the UAVs are not point masses, there are communications delays, packets are lost, and the simulation includes wind. When comparing the different optimiztions, on 8 of the 32 batches the actual system did complete faster than the other "optimal"solution. Equally interesting is the outlier. During this batch the tasks were tightly clustered, forcing the majority of the execution time to be spent turning. The optimal solution does not take the turning into account and thus the actual system appears to perform very poorly.

VI. Conclusion

CSL, a high-level feedback control language for mobile sensor networks, was informally explained. CSL was shown to enable the specification of reactive network missions. The CSL Execution Engine automatically allocates resources to accomplish these user defined network objectives. This allows users to concentrate on what information they desire and not on the specifics of how to obtain it.

CSL's implementation was shown to add insignificant overhead to the execution time required to complete the objectives. The non-optimality of the performance was shown to stem from the inability to solve the NP-hard open traveling salesman probem efficiently during execution.

CSL also emphasizes and proposes adaptability through patching. As new information is available from the network, the user may modify the network objectives without interrupting the execution. This enables heirarchical control, error correction, and the ability to adapt to unexpected events within the mobile sensor network.

Current research efforts include extending the language to multiple users; the web services already accept multiple connections but permissions need to be developed. Extending the conditions to allow external functions to read the CSL state and produce boolean condition values is also of high priority. Finally, the distribution of the Publisher component is being investigated to create more of a peer-to-peer network interaction.

Acknowledgment

The Authors would like to acknowledge the contributions of Eduardo Marques for his feedback during the development of the Run-time Patching Language.

References

[1] C. Duarte and B. Werger, "Defining a common control language for multiple autonomous vehicle operation," in *OCEANS MTS/IEEE Conference and Exhibition*, vol. 3, 2000, pp. 1861–1867.

[2] S. Mupparapu, S. Chappell, R. Komerska, D. Blidberg, R. Nitzel, C. Benton, D. Popa, and A. Sanderson, "Autonomous systems monitoring and control (asmac) - an auv fleet controller," in *Autonomous Underwater Vehilces*, 2004, pp. 119–126.

[3] M. Hsieh, L. Chaimowicz, A. Cowley, B. Grocholsky, J. Keller, V. Kumar, C. Taylor, Y. Endo, R. Arkin, B. Jung, D. Wolf, G. Sukhatme, and D. MacKenzie, "Adaptive teams of autonomous aerial and ground robots for situational awareness," *Journal of Field Robotics*, 2007.

[4] Y. Kanayama and C. Wu, "It's time to make mobile robots programmable," in *Proc. International Conference on Robotics and Automation (ICRA)*, vol. 1, 2000, pp. 329–334.

[5] D. Duhaut and E. Manacelli, "Including control in the definition of a programming language for multi-robots," in *IEEE/RSJ International Workshop on Intelligent Robots and Systems (IROS)*, vol. 3, 1991, pp. 1382–1387.

[6] D. Mackenzie, R. Arkin, and J. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, March 1997.

[7] H. Nishiyama, H. Ohwasa, and F. Mizoguchi, "A multiagent robot language for communication and concurrency control," in *International Conference on Multi Agent Systems*, 1998, pp. 206–213.

[8] L. Xu and U. Ozguner, "Battle management for unmanned aerial vehicles," in *Proceedings of 42nd IEEE Conference on Decision and Control*, vol. 4, 2003, pp. 3585–3590.

[9] C. Kirsch, L. Lopes, and E. Marques, "Semantics-preserving and incremental runtime patching of real-time programs," in *Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2008.

[10] G. Baliga, S. Graham, C. Gunter, and P. Kumar, "Reducing risk by managing software related failures in networked control systems," in *Proceedings of the 45th IEEE Conference on Decision and Control*, 2006, pp. 2866–2871.

[11] M. Usher and D. Jackson, "A concurrent visual language based on petri nets," in *IEEE Symposium on Visual Languages*, 1998.

[12] S. Vinoski, "Concurrency with erlang," *IEEE Internet Computing*, vol. 11, no. 5, pp. 90–93, Sept-Oct 2007.

[13] A. Deshpande, A. Göllü, and P. Varaiya, "Shift: A formalism and a programming language for dynamic networks of hybrid automata," *Hybrid Systems*, pp. 113–133, 1996.

[14] S. Baranov, "Synthesis of control units for mobile robots," in *Second EUROMICRO workshop on Advanced Mobile Robots*, 1997, pp. 80–86.

[15] I. Schiller and J. Draper, "Mission adaptable autonomous vehicles," in *IEEE Conference on Neural Networks for Ocean Engineering*, 1991, pp. 143–150.

[16] M. Lundell, J. Tang, and K. Nygard, "Fuzzy petri net for uav decision making," in *International Symposium on Collaborative Technologies and Systems*, 2005, pp. 347–352.

[17] A. Ryan, J. Tisdale, M. Godwin, D. Coatta, D. Nguyen, S. Spry, R. Sengupta, and J. Hedrick, "Decentralized control of unmanned aerial vehicle collaborative sensing missions," in *American Control Conference (ACC)*, 2007, pp. 4672–4677.

[18] J. Tisdale, A. Ryan, M. Zennaro, X. Xiao, D. Caveney, S. Rathnam, R. Sengupta, and J. Hedrick, "The software architecture of the berkeley uav platform," in *IEEE Control Systems Society Conference*, 2006.