Abstract—The design of a distributed control framework for teams of unmanned combat air vehicles (UAV) is discussed. The problem is formulated for a scenario where there is a set of UAVs with different capabilities (sensors, weapons, decoys), collectively called the Blue force, and a set of targets called the Red force. The Blue force is used to attack the Red force, which threatens the attacking Blue force. The distributed control framework consists of a hierarchy of task controllers, vehicle supervisors, and elemental maneuver feedback controllers, which determine actual flight paths, weapons release, and sensor management. The control framework is described as interacting hybrid automata using Shift a programming language for dynamic networks of hybrid automata.

I. INTRODUCTION

We present a framework for the mixed initiative coordination and control of teams of UAVs. The ‘mixed initiative’ component of the framework allows human operators to guide, and override, the specification, command, and supervision of complex UAV operations in hostile environments (see [1] for a description of missions and related technological issues).

We consider the following setting: there is a set of UAVs with different capabilities (sensors, weapons, decoys), collectively called the Blue force, and a set of targets called the Red force; The Blue force is used to attack the Red force, which threatens the attacking Blue force.

The design space of Blue’s attack is large and heterogeneous involving the selection of targets and the order in which they are attacked; equipping and assigning UAV teams to these targets; coordinating the attack so that the order of the attack is maintained; re-allocating UAVs to teams to cope with uncertainty and attrition endured by these teams; and then designing real-time feedback strategies for UAV flight and sensor and weapons use. We structure the design in two layers: off-line plan, and online execution control.

The off-line planning procedure uses a ‘mixed initiative’ planning tool to assist human operators to select sequences of targets and risk minimizing paths. The plan consists of a list of tasks, which are composed of several sub-tasks, of precedence relations among sub-tasks, and of an assignment of UAV teams to tasks. Each sub-task is composed of several legs. Each leg consists of an attack to a specific target or of a path to a given destination. There is a partial order for the legs composing a task. The legs composing a sub-task satisfy a total order.

Online execution is a distributed control problem, where information and commands are exchanged among multiple vehicles, and the roles, relative positions, and dependencies of those vehicles, as well as their numbers, change during operations in a hostile environment (see [7] for a survey on distributed control of hybrid systems). The concepts for execution control build on our experience in the design of distributed control hierarchies described in [6], [5], [2], [3].

The design challenges are: 1) to organize a control structure to control plan execution with high levels of performance in the presence of hostile behavior from the Red force; 2) to coordinate the spatial and inter-temporal utilization of a large number of UAVs (10 to 100 vehicles) in a large geographic area (hundreds to thousands of square kms), on several time-scales (minutes, hours, days); 3) to abstract the key aspects of operations and performance of teams to reduce the complexity of the problem while maintaining sub-optimal levels of performance; and 4) to allow intervention by experienced human operators.

The structure of the plan (legs, sub-tasks and tasks) lends itself to a simple decomposition of control problems: 1) UAVs attack in successive waves along shared attack paths to minimize exposure to risk (only the UAVs engaged in a specific attack are exposed to risk while the others keep wait and advance when it is their turn to conduct attacks); and 2) individual UAV behavior can be abstracted by a few action/motion templates also called maneuvers. We designed the distributed control hierarchy using structuring concepts. At the bottom layer there are UAV platforms. The next layer concerns single vehicle control and comprises the following components: a vehicle supervisor to interact with external entities and to command the execution of maneuvers; maneuver controllers to execute them; and a dispatcher to supervise plan execution for single UAV operations. The vehicle supervisor has a list of links to external controllers, in most cases team controllers. External controllers or human operators can change these links to transfer the UAV from one team to another one; the UAV operates independently in the absence of such links. The top layer, the team control layer, comprises task and sub-task controllers. Task controllers coordinate the concurrent execution of the constituent sub-tasks by different teams of vehicles. This is done with the help of a few coordination variables describing the aggregate state of each team.

We model this control hierarchy in Shift, a modelling language to describe dynamic networks of hybrid automata [4]. Shift provides a formal syntax and semantics that incorporates
first order predicate logic to allow hybrid automata to interact through dynamically reconfigurable input/output connections and synchronous composition. These constructs are particularly useful to model and control teams of vehicles, and also provide for an operational definition of a team: a team is a set of vehicles, controllers, and links among them. In this setting, we understand a link not as a fixed part of the system, but as a control variable that we can manipulate.

The paper is organized as follows. In section II we present an aside on Shift, the simulation language we use to model our control structure. In section III we discuss the problem domain and task specification. In section IV we describe our architecture for the distributed control of teams of UAVs. In section V we present a description of the controllers in this hierarchy. In section VI we draw the conclusions.

II. AN ASIDE ON Shift

We model our design in Shift to encode the essential features of the distributed control architecture. Each model in the architecture is a Shift component (these are classes in the language of object-oriented programming). Each Shift component has a well-defined interface to facilitate combining controllers in the hierarchy. Figure 1 illustrates what these components look like and how they interact. A component description includes a data model (inputs, outputs, and internal state), and hybrid dynamics (discrete state or mode changes and differential equations). Instances of components have unique names. Shift provides constructs to compose components by synchronous composition of discrete transitions, by input-output connections, and by dynamically linking different components. These constructs use the names of components or first-order predicate logic.

![Fig. 1. A Shift component.](image)

III. THE PROBLEM

A. Task specification

There is a set of UAVs with different capabilities (sensors, weapons, decoys), collectively called the Blue force, and a set of entities (trucks, surface-to-air missiles, etc.) called the Red force, which threatens the attacking Blue force. The Red force has entities specialized to protect it against UAVs. There are several sources of uncertainty. Examples include the number and locations of targets, and the state of the targets after an attack.

The activities of the Blue force are organized as tasks. There are several types of tasks, e.g. surveillance, attack. Here, we focus on attack tasks. The initial specification for an attack task consists of a selection of entities from the Red force called the primary targets. The primary targets may be defended by other entities from the Red force. In order to reduce the threat level endured by the attacking UAVs it may be necessary to attack first a selection of the defending entities. These are called secondary targets and are added to the initial task specification.

Throughout this paper we assume that we are given a task specification, which includes a team of UAVs to execute it, generated by the off-line planning procedure. The issues of task specification become more clear with the example represented in figure 2: there are 3 primary targets, $P_1, P_2$ and $P_3$, and several secondary targets $S_1, \ldots, S_5$.

![Fig. 2. Task specification.](image)

This task consists of three sub-tasks. Each sub-task consists of sequence of legs. Each leg represents an attack to a specific target (primary or secondary) or a path to a given destination. The 3 sub-tasks are to be executed concurrently. There are precedence relations on the order of execution of the legs: there is a partial order on the task legs and a total order on the sub-task legs. For example leg$_0$ from sub-task$_2$ precedes leg$_4$ from sub-task$_1$.

This partial order arises from the structure of the risk minimizing strategies produced by the off-line planning procedure. This is because of the structure of the risk function. In the simplest risk models the risk posed by an individual Red force entity is modelled as the indicator function$^1$ of a circle centered at the location of the entity – and does not account for directionality, as is the case with more elaborate models. Depending on the type of air defenses, the contributions of individual risk functions are either additive or taken as their

$^1$The indicator function for a set is 1 for all points belonging to the set and 0 outside.
maximum. Risk minimizing strategies eliminate targets in a sequence designed to open safe corridors to the primary targets. This is the reason why motions proceed along shared paths and why there is a total order for leg execution of a sub-task. The partial order for task execution arises from the interfering effects of the concurrent creation of multiple penetration routes to the primary targets.

Formally, a Task specification is a pair: $\text{task} = \{(\text{SubtaskList}, \geq), (\text{TeamList}, \text{assign})\}$ where:

- $\text{SubtaskList} = \{\text{subtask}_1, \ldots, \text{subtask}_n\}$ is a set of sub-tasks, each of which is an array of legs, $\text{subtask}_i = \{\text{leg}_{i,1}, \ldots, \text{leg}_{i,n}\}$. There is a partial order $\geq$ on the legs composing a task. The legs composing a sub-task satisfy a total order. However, there is a partial order on the task legs.

- $\text{TeamList} = \{\text{team}_1, \ldots, \text{team}_n\}$, and $\text{assign} : \text{TeamList} \rightarrow \text{SubtaskList}$ is an assignment (or 1-1 function) of teams to sub-tasks.

Figure 3 presents $\text{Shift}$ skeletons for the task, sub-task, and leg components. These are basically data models. The task component consists of an array of sub-tasks defined as the output variable $s$. The sub-task component encodes the assignment of teams to sub-tasks, defined as the output variable $\text{team}$ of type set($\text{ucav}$), and the array of legs composing the sub-task, defined as the output variable $p$, an array of legs. The leg component encodes the leg specifications as output variables: $p$\_attack – the path specifications, an array of numbers; platform target – target (which may be nil), of type em platform; requires – preceding legs, a set of legs; and additional directives for execution.

<table>
<thead>
<tr>
<th>type task</th>
</tr>
</thead>
<tbody>
<tr>
<td>( output array(subtask) s; )</td>
</tr>
<tr>
<td>type subtask</td>
</tr>
<tr>
<td>( output array(leg) p; set(ucav) team; ... )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type leg</th>
</tr>
</thead>
<tbody>
<tr>
<td>( output array(array(number)) p_attack; platform target; set(leg) requires; mspec wpn = hold; mspec wpc = hold; ... )</td>
</tr>
</tbody>
</table>

Fig. 3. Left is a skeleton of task and sub-task components; right is a skeleton of a leg component.

### B. Execution concepts

The task specification basically defines execution constraints, namely a partial order for task execution. There may exist multiple online control policies satisfying these constraints. This provides for additional flexibility in execution and mixed initiative operations. As an example consider the following two execution policies for the task represented in figure 2: 1) for each sub-task dispatch UAVs one by one – dispatch a new UAV when the previous one runs out of bombs or is eliminated; or 2) send a team of UAVs flying together over safe paths and send one at a time to attack targets while the others wait in a safe location.

The concepts for execution control build on our experience in the modular design of distributed control hierarchies described in [6], [5], [2], [3]. In previous designs we have used the concept of maneuver – a prototype of an action/motion description for a single vehicle – as the atomic component of all execution concepts. This is also a suitable abstraction of the operations required to execute attack specifications.

The maneuvers required to execute the attack task specification are: follow\_path – path following maneuver; attack\_target – attack maneuver that maximizes the probability of destruction of the target while minimizing the accumulated risk; and hold – a holding pattern. The attack maneuver encodes the attack logic: it concerns the control of weapons and risk reduction devices, such as jammers, and path optimization and execution procedures. This is done in a feedback manner.

There are two types of legs in the attack task specification: attack and path. An attack leg starts with a safe path maneuver (the path may be empty). By safe we mean risk free. Next, it may be followed by a holding pattern in preparation for the third maneuver. The third maneuver is either an attack or a follow path (safe path) maneuver depending on the state of the target. This maneuver may be followed by another holding pattern. This is why the $\text{Shift}$ skeleton of the leg component includes the specifications for the attack maneuver and for the first safe path, and the specifications for the two holding maneuvers – $\text{mspec wpn = hold}$ and $\text{mspec wpc = hold}$. The specification for a path leg does not include a target. It consists of a follow path maneuver (safe path) that may be followed by a holding maneuver.

This leg specification allows for several execution policies: the execution sequence depends on the type of the leg, on the state of the target (enabled or disabled), and on the execution control policy.

We now describe one online execution control policy to illustrate the main execution concepts. We will be referring to this policy for the reminder of the paper.

Execution proceeds as follows. Each sub-task is executed by a team of UAVs. The team is organized as two sub-teams: $\text{attacker}$ and $\text{reserve}$. Initially, the $\text{attacker}$ sub-team is empty; the $\text{reserve}$ sub-team receives the team of vehicles allocated to the sub-task. Execution starts with the first leg of the $\text{subtask}$. The $\text{reserve}$ vehicles execute a follow\_path maneuver. When the end of the path is reached one of the $\text{reserve}$ UAVs is transferred to $\text{attacker}$ and the two sub-teams start two concurrent threads of execution until the $\text{subtask}$ terminates successfully or fails. The $\text{attacker}$ UAV leads the execution: it executes the $\text{sub-task specification}$ until successful termination, or until it is destroyed or runs out of bombs – in these two cases the UAV is removed from the sub-team $\text{attacker}$ and another UAV from $\text{reserve}$ is tasked to replace it. The $\text{reserve}$ sub-team follows the $\text{attacker}$: it moves
to the farthest safe point of the subtask legs terminated so far by the attacker; when this point is reached they execute a holding maneuver until this point can be moved further as a result of the actions of the attacker.

This execution policy minimizes both the total risk incurred by the team and the time required to execute the task.

The interpretation of an attack task can be further specialized. For example, it may be necessary to use UAVs with damage assessment capabilities to evaluate the state of each target before execution proceeds to the next leg. This requires an additional sub-team, bda, and the coordination of its actions with those of attacker and reserve. This is done easily in our on-line execution framework.

Our problem consists of designing a structure of controllers to execute attack task specifications according to some online control policy.

IV. CONTROL ARCHITECTURE

We derived the distributed control hierarchy for task execution from the specification and execution concepts discussed in the previous section. This control architecture is depicted in figure 4. The key concepts are described next:

Platform. A ‘mirror’ of the vehicle platform and the interface to the vehicle hardware.

Maneuver controller. Supervises the execution of a maneuver (there is a library of maneuver controllers). It sends commands to the platform and gets the platform status. It accepts abort and configuration commands from the vehicle supervisor and sends status messages to it. It is created by the vehicle supervisor and deletes itself when the maneuver terminates. There is always one active maneuver controller in the UAV.

Vehicle supervisor. Supervises the UAV operations. It receives maneuver specifications through a link to either the dispatcher – in autonomous mission execution – or to a team controller – in team task execution – and launches the corresponding maneuver controller and monitors its execution and the state of the vehicle. It also accepts configuration commands from an external controller. This makes it possible to move the UAV from one team to another just by changing the link to the team controller. The vehicle supervisor is the same throughout the life span of the UAV.

Vehicle dispatcher. Supervises the execution of an autonomous mission. This is used when the UAV has to execute a mission independently and autonomously. Basically the dispatcher repeats the following pattern of interactions: gets the next maneuver specification from the mission, sends it to the vehicle supervisor for execution, and waits for its completion.

UAV. The UAV unit. It is composed of a platform, the vehicle supervisor, and the dispatcher. It interacts with external entities, such as a team controller, through the vehicle supervisor.

Team controller. Supervises the execution of a task specification. Basically it commands and monitors the execution of UAV maneuvers to execute the task specification. It does this by exchanging messages and commands with the vehicle supervisors in the team. The team controller also accepts configuration and abort commands.

The overall control structure evolves as time progresses:
1) we create task and sub-task controllers to execute a task, and delete them when the corresponding task or sub-task is completed; 2) each sub-task controller is linked to two sub-teams of UAVs to coordinate sub-task execution; 3) the links change with time as UAVs are transferred to and from sub-teams, or are eliminated.

This control architecture, and its Shift implementation, allow for the introduction of additional layers on the top of the existing ones. This allows for more complex patterns of interactions that may be required to model more complex operations.

V. CONTROLLERS

In this section we describe the implementation of the controllers in this distributed control architecture with reference to the execution policy described in section III.

The Team controller for the attack task consists of a task controller and a set of sub-task controllers, one per sub-task. These controllers are linked through input-output connections: each sub-task controller has a link to its task controller, and the task controller has a link to each sub-task controller. We use these links to exchange commands, messages and data among these controllers.

A. Vehicle controllers

The vehicle supervisor supervises and controls the UAV. It accepts commands from either the dispatcher or from a sub-task controller.

The dispatcher maintains the state of execution of a mission – the index of the last maneuver executed successfully – and when it receives the done event from the vehicle supervisor updates this index and commands the vehicle supervisor to execute the next maneuver specification.
The \textit{Shift} skeleton of the vehicle \textit{supervisor} is depicted in figure 5. It has the following input variables (which can be changed by an external entity): \textit{vs} – a link to a set of control centers; \textit{tc} – a link to a task controller; \textit{stc} – a link to a sub-task controller; \textit{op} – a link to human operator; and \textit{peers} – a set of links to other UAVs to allow for direct communication. The output variables (which are available to external entities) are: \textit{aserv} – the array of services provided by the UAV; \textit{am} – the set of maneuvers it is able to execute; \textit{im}, \textit{om} – the input and output messages it accepts. The state variables (describing the internal operation of the supervisor) are: \textit{mc} – a link to the current maneuver controller; \textit{is} – a link to the internal systems; \textit{vs} – the state of the UAV; \textit{es} – the set of services currently provided by the UAV; and \textit{ds} – a link to the dispatcher. The discrete states are: \textit{Init}, \textit{Exec}, \textit{Error}, and \textit{Idle}. Each transition is labelled with a guard, the condition under which the transition can take place, and an event, the message sent when the transition is taken. The transition structure, which is not depicted in figure, 5, is briefly described next. The vehicle supervisor is initially in the state \textit{Idle}. Upon the reception of a maneuver specification it creates a maneuver controller if all of the vehicle systems are GO. Otherwise, the transition to the \textit{fail state} is taken, and it sends an \textit{error(code)} event to the plan supervisor, and the UAV fails.

\begin{verbatim}

type supervisor {
    input // what we feed to it
    set(controlcenter) cs;
    team_controller tc;
    dispatcher ds;
    operator op;
    set(UAV) peers;

    output // what we see from outside
    array[services] aserv;
    set(maneuver) am;
    input_messages im;
    output_messages om;

    state // what’s internal
    maneuver_controller mc;
    internal_systems is;
    state_vehicle vs;
    array(service) es;

    discrete // discrete modes
    Init, Exec, Err, Idle;
}
\end{verbatim}

Fig. 5. Skeleton of the vehicle supervisor.

There is a hierarchy of maneuver controllers. The root component of this hierarchy is sketched in figure 6. The root \textit{elemental_maneuver} has 5 states: \textit{Init}, \textit{Exec}, \textit{Error}, \textit{Done}, and \textit{Stop}. It enters the state \textit{Init} immediately after being created by the \textit{vehicle supervisor}. The transition to the \textit{Exec} state is taken immediately if the the vehicle systems are GO. If not, the transition to the \textit{Error} state is taken and the event \textit{error(code)} is sent to vehicle supervisor. In the \textit{Exec} state, the vehicle supervisor executes a pre-specified holding pattern (a rectangle or a circle) for a given period of time \(T\). At the end of this period the transition to \textit{Done} is taken, the event \textit{done} is sent to the vehicle supervisor. Then, the transition to \textit{Stop} is taken and the \textit{maneuver controller} is deleted. The \textit{maneuver controller} accepts an abort command from the \textit{vehicle supervisor} or from the operator in any of its states. If this is the case it takes the transition to \textit{Stop} immediately.

\begin{verbatim}

type root_elemental_maneuver {
    input
    state_vehicle vs;
    maneuver_spec mspec;

    output
    array[number] actuatorcommands;
    symbol t;

    state
    set[services] requires;
    ...
}
\end{verbatim}

Fig. 6. Skeleton of the root Maneuver Controller.

\subsection{Task controller}

The left box in figure 7 depicts the \textit{Shift} skeleton of the \textit{task controller}. It has one input variable \textit{t}, the \textit{task specification}, one output variable \textit{st}, the set of \textit{sub-task controllers}, and two state variables, \textit{task Legs done} and \textit{fail}. \textit{task Legs done} is the set of \textit{leg} done so far. \textit{fail} is a flag to signal that at least one of the sub-tasks failed. The task fails if one of the sub-tasks fails.

The task controller coordinates the concurrent execution of all sub-tasks with the corresponding sub-task controllers. The coordination variables are \textit{task Legs done} and \textit{fail}. The coordination mechanism is quite simple. Each sub-task controller updates and accesses these variables as follows: checks \textit{task Legs done} for precedence constraints on a leg before starting its execution; checks \textit{fail} to determine when to enter a fail mode (in this mode the UAVs are commanded to move to the closest safest point and enter a holding pattern when they arrive there); writes the fail condition to the variable \textit{fail} when the sub-task fails; and upon successful completion of the current leg adds it to the set \textit{task Legs done}.

\subsection{Sub-task controller}

The \textit{sub-task controller} models a novel coordination strategy for interacting teams. The right box of figure 7 depicts the \textit{Shift} data model for the \textit{sub-task controller} component. This component controls and coordinates the concurrent operations of two sub-teams of UAVs: \textit{attacker} and \textit{reserve}. The two
The input variables are: \( p \) – the sub-task specification; and \( c \) – a link to the task controller. The output variables are: reserve – the set of reserve UAVs; executed – the set of legs executed so far by attacker; and accept – an enabling condition. The state variables are: attacker – the sub-team in charge of attack; vsa and vsr – sets of links to the vehicle supervisors in the attacker and in the reserve sub-teams respectively; current\(_a\)_leg and current\(_r\)_leg – the legs being executed by the attacker and the reserve sub-teams respectively; attack\(_a\)_stage and reserve\(_a\)_stage – the current stage of execution of current\(_a\)_leg and current\(_r\)_leg respectively (each leg is a sequence of maneuvers); and preceeding\(_a\)_legs – the set of precedence constraints for the next attack leg.

**Interactions with UAV vehicle supervisors.** The two sets of vehicle supervisors (vsa and vsr) model links to the individual UAV supervisors in each sub-team. We use predicates on the state of the UAVs in each sub-team to access the corresponding vehicle supervisors. We use this construction to transfer UAVs (and control authority) between the 2 sub-teams. The construction is robust with respect to the elimination of UAVs. Consider the following sequence of events to clarify the properties of this construction.

1) the attacker UAV is eliminated (or has to leave the sub-team for some other reason);
2) the sub-task controller removes the corresponding supervisor from vsa, removes the UAV from attacker, and updates current\(_a\)_leg to the value of current\(_r\)_leg;
3) if reserve is empty the sub-task fails; if not the sub-task controller transfer one UAV from the reserve sub-team to the attack sub-team; the corresponding supervisor is also transferred from vsr to vsa.

The **Shift code** for the actions described in 2) is as follows:

```shift
    allocate_attacker -> execution (vs:execute),
```

The self-transition from the **execution** state to itself takes place when there is at least one UAV in reserve. In this case, the UAV that is closest to the target destination is transferred to attackers and the corresponding variables updated.

**Coordination variables.** The interactions between the two teams are governed by a few coordination variables that abstract away the execution details while providing the information required to make decisions. The coordination variables are: attacker and reserve; current\(_a\)_leg and current\(_r\)_leg; attack\(_a\)_stage and reserve\(_a\)_stage; and preceeding\(_a\)_legs. The variables attack\(_a\)_stage and reserve\(_a\)_stage maintain the execution state of the current\(_a\)_leg and current\(_r\)_leg legs for the attacker and reserve sub-teams respectively.

The attacker sub-team is in one of the following leg execution states: attack (attack segment of a leg); path (safe path segment); and hold (holding pattern at the end of the leg). The reserve sub-team is in one of the following states: hold\(_end\) (holding pattern at the end of the leg), hold\(_path\) (holding pattern at the end of the safe path), path (executing the safe path of the leg), and path\(_attack\) (executing the attack path of the leg).

**Transition structure.** The sub-task controller hybrid automaton has three states: **execution**, **initialize**, **error**. There is basically one state **execution** where the normal operation of the controller takes place. The other states are either preparation states (**initialize**) or error and fail states. The transition structure is defined by predicates on the values of the values of the coordination variables. This provides for a high level of abstraction and for a more compact notation. The majority of the transitions are self-loops for each discrete state, in particular for the **execution** state. The other transitions
concern transitions among the discrete states. Mathematically, this is equivalent to defining the transition structure on a larger set of discrete states. This style is more convenient for programming since the behaviors of attacker and reserve can be specified independently (with some coupling resulting from precedence relations on attack and reserve legs) and transitions can be easily added and deleted without additional changes on the remaining code. This style also allows for a more convenient abstraction of the configuration structure and can be easily extended to accommodate interactions with other teams of UAVs.

The actions on the sub-task controller transition system consist in one of the following: 1) command maneuver execution to vehicle supervisors in vsa or/and in vsr; 2) transfer vehicles from reserve to attacker when the second sub-team becomes empty; 3) remove a UAV from reserve and/or attacker when the UAV is destroyed or it has to leave the team for some other reason. Some of these actions result in commands sent to one UAV supervisor, others result in commands sent to the supervisor of a sub-team. Next, we include illustrative Shift examples.

The situation where the sub-task controller removes the attacker UAV from the sub-team attacker when this UAV runs out of bombs after attacking a target which is not the last target in this sub-task \((a\text{\_step} < \text{steps}(p))\) is coded as follows:

```plaintext
execution -> execution {vsae\_nobombs(one:p)} when (a\_step < \text{steps}(p))

define
{
  ucap v:=u(p);
  supervisor vsp:=p;
}
do
{
  vsa := vsa - (p);
  attackers := attackers - (u(p));
  a\_step := r\_step;
  current\_a\_leg := current\_r\_leg; // leg specs
  attack\_stage := $path;
  tc(vsp) := nil;
},
```

Note that \(p\) gives the vehicle supervisor for the UAV that ran out of bombs. This local variable enables us to access the corresponding ucap \(v\), and to set the team controller link in \(p\) to nil to remove the corresponding UAV from this team.

The following excerpt of Shift code models the case where the task fails and all the supervisors vsr are commanded to abort the current maneuver under execution.

```plaintext
execution -> execution {vsae\_nobombs(one:p)} when (fail(task_c)=1)

... \(\ldots\)
```

D. Shift example

Figure 8 presents a Shift specification for a scenario such as the one described in section III. There are 4 UAVs, one task composed of two sub-tasks, and the corresponding task and sub-task controllers. The Shift code illustrates how to create a complete simulation. This is done in several steps (transitions of the hybrid automaton task\_simulation) to respect the creation dependencies. The last two steps create the task and the sub-task controllers.

![Fig. 8. Shift specification for a specific scenario.](image)

VI. CONCLUSIONS

This control architecture and its Shift implementation are a great advance over current practice in the specification of controller design. The specification takes advantage of the object-oriented nature of Shift and of the dynamic configuration constructs available in Shift. In particular:

- Specification of controllers is separated from their instantiation;
- Controllers are hierarchically organized;
- Task and sub-task controllers operate on a varying number of UAVs;
- Controllers are created and removed during operations to accommodate the evolving nature of UAV operations in a hostile environment.
- User intervention is explicitly made available at all levels of the hierarchy in terms of a well-defined interface of command and response messages;
Controllers can be extended through specialization (because Shift allows inheritance).

The structure of the task and sub-task controllers encodes a framework for multi-team coordination and control. The control actions are based on the values of a few coordination variables describing the state of each team and of each sub-team. This provides for several levels of description and aggregation, thus facilitating mixed initiative interactions where the operator has to interact with multiple vehicles and teams whose structure changes with time.

Vehicle and team controllers can be easily expanded in this online execution framework. Currently teams only carry out an attack task. But if other task types, such as search or jamming, are implemented, the team controller can be augmented in a modular fashion to accommodate these. Similarly, the vehicle controller can be augmented to include other maneuvers or tactics.

The structure of the task specification allows for a compact representation that is interpreted differently according to the state of execution, to the type of abstraction utilized, and to the selected control policy.

ACKNOWLEDGMENTS

Research supported by Darpa Contract F33615-01-C-3150.

REFERENCES


