Complex Task Allocation in Mixed-Initiative Delegation: A UAV Case Study (Early Innovation)

David Landén*
Linköping University
581 83 Linköping, Sweden
david.landen@liu.se

Fredrik Heintz
Linköping University
581 83 Linköping, Sweden
fredrik.heintz@liu.se

Patrick Doherty
Linköping University
581 83 Linköping, Sweden
patrick.doherty@liu.se

ABSTRACT

Unmanned aircraft systems (UASs) are now becoming technologically mature enough to be integrated into civil society. An essential issue is principled mixed-initiative interaction between UASs and human operators. Two central problems are to specify the structure and requirements of complex tasks and to assign platforms to them so that they can be achieved. We have previously proposed task specification trees (TSTs) as a highly expressive specification language for complex multiagent tasks that supports mixed-initiative delegation with adjustable autonomy. The main contribution of this paper is a sound and complete distributed heuristic search algorithm for allocating the individual tasks in a TST to platforms. The allocation also instantiates the parameters of the tasks such that all the constraints of the TST are satisfied. Constraints can be used to model dependencies between tasks, resource usage as well as temporal and spatial requirements on the complex task. Finally, we discuss a concrete case study with a team of unmanned aerial vehicles assisting in a challenging emergency services scenario.

1. INTRODUCTION

Unmanned aircraft systems (UASs) are now becoming technologically mature enough to be integrated into civil society. Principled interaction between UASs and human resources is an essential component in the future uses of UASs in complex emergency services scenarios. Mixed-initiative interaction between human operators and such systems will be central. By mixed-initiative, we mean that interaction and negotiation between a UAS and a human will take advantage of each of their skills, capacities, and knowledge in developing a mission plan, executing the plan, and adapting to contingencies during the execution of the plan. In developing a principled framework for such sophisticated interaction in complex scenarios, a great many interdependent conceptual and pragmatic issues arise and need clarification both theoretically and pragmatically in the form of demonstrators.

Two central problems are to define complex mixed-initiative missions and given a mission find platforms which together can execute it. We have previously proposed task specification trees (TSTs) as a highly expressive specification language for multiagent tasks that supports mixed-initiative delegation with adjustable autonomy [4]. A task is recursively defined as a tree of tasks where temporal requirements and interdependencies among tasks are specified as constraints. In this paper we describe an allocation algorithm for TSTs together with a concrete unmanned aerial vehicle (UAV) case study. The allocation algorithm assigns platforms to tasks and instantiates the parameters of the tasks such that all the constraints of the TST are satisfied. The algorithm recursively searches among the potential allocations in a distributed manner and uses distributed constraint satisfaction techniques to check if an allocation satisfies the task and platform constraints. The case study gives a detailed example of the allocation algorithm applied to a team of UAVs assisting in a challenging emergency services scenario involving delivery of food and medical supplies to injured people.

2. THE DELEGATION FRAMEWORK

To support cooperative goal achievement among a group of agents a delegation framework has been developed [4, 5]. It provides a formal framework for describing and reasoning about what it means for an agent to delegate an objective, which can be either a goal or a plan, to another agent. The concept of delegation allows for studying not only cooperation but also mixed-initiative problem-solving and adjustable autonomy.

By delegating a partially specified objective the delegee is given the autonomy to complete the specification itself. By making the objective more specific the autonomy is limited. If the delegated objective is completely specified then the agent has no autonomy when it comes to achieving the objective. By allowing both agents and human operators to partially specify an objective, mixed-initiative problem-solving is supported.

2.1 Task Specification Trees

A *task specification tree* (TST) is a distributed data structure with a declarative representation that describes a com-

*This work is partially supported by grants from the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR), the VR Linnaeus Center CADICS, the ELLIIT Excellence Center at Linköping-Lund for Information Technology, and the Center for Industrial Information Technology CENIIT.

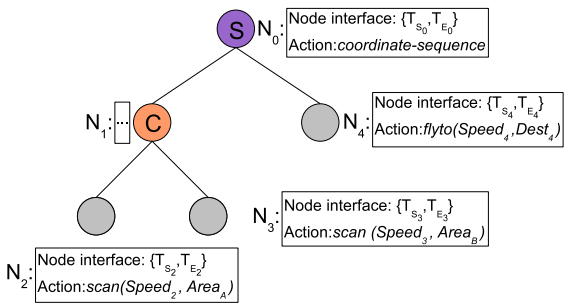

Figure 1: An example TST for first scanning $\mathsf{Area_A}$ and $\mathsf{Area_B}$ concurrently and then flying to $\mathsf{Dest_4}$.

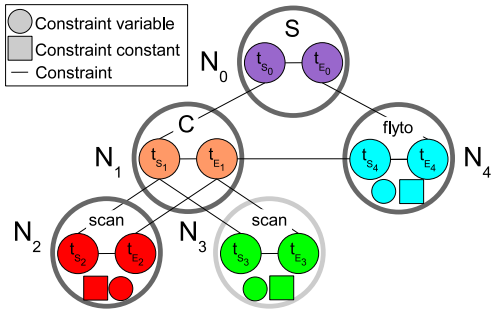

Figure 2: The constraint network defined by the TST in Figure 1.

plex multiagent task. Each node in a TST corresponds to a task that should be performed. Each node has a *node interface* consisting of a set of parameters called *node parameters* that can be specified for the node. The node parameters determine task specific details of the node.

Nodes in a TST either specify actions or goals. Actions can either be primitive or composite. A *primitive action* is a leaf node in the TST while a *composite action* is an interior node. Action nodes can be executed when instantiated, whereas goal nodes first require a plan to be generated. The plan then becomes a new TST branch that in turn can be instantiated and executed. A TST without any goal nodes is called *fully expanded*. Nodes can also be removed and added during execution, for example, to repair a TST after a failure. When a TST has been executed, the resulting TST represents the history of the mission, including concrete task instantiations, errors, and repairs.

Figure 1 shows an example TST for first scanning $\mathsf{Area_A}$ and $\mathsf{Area_B}$ concurrently and then flying to $\mathsf{Dest_4}$. Nodes N_0 and N_1 are composite action nodes, sequential (S) and concurrent (C), respectively. Nodes N_2, N_3 and N_4 are primitive action nodes. Each node specifies a task and has a node interface containing node parameters. In this case only temporal parameters are shown representing the respective intervals a task should be completed in.

Each node can have constraints associated with it, called *node constraints*. These constraints limit the valid values of the node parameters. A TST can also have *tree constraints*, expressing precedence, dependence, and organizational relations between the nodes in the TST. Together the node parameters and the constraints form a constraint network. Setting the value of a parameter constrains not only the network, but implicitly, also the degree of autonomy of an agent. Figure 2 shows the constraint network defined by the TST in Figure 1.

3. ALLOCATING TST SPECIFIED TASKS

Given a TST representing a complex task, an important problem is to find a set of platforms that can execute these tasks according to the specification. The problem is to allocate tasks to platforms and assign values to parameters such that each task can be carried out by its assigned platform and all the constraints of the TST are satisfied.

For a platform to be able to carry out a task, it must have the *capabilities* and the *resources* required for the task. A platform that can be assigned a task in a TST is called a *candidate* and a set of candidates is a *group*. The capabilities of a platform are fixed while the available resources will vary depending on its commitments, including the tasks it has already been allocated. The resources and the commitments are modeled with constraints. Resources are represented by variables and commitments by constraints. The resources used by a platform when executing a particular action are represented by a parameterized set of constraints. The action parameters must be part of the node interface for any node containing that action. These constraints are local to the platform and different platforms may have different constraints for the same action. Figure 3 shows the constraints for the scan action for platform P_1.

When a platform is assigned an action node in a TST, the constraints associated with that action are instantiated and added to the constraints of the platform. The platform constraints are connected to the constraint problem defined by the TST through the node parameters in the node interface. Figure 4 shows the constraint network after allocating node N_2 from the example TST to platform P_1.

A platform can be allocated more than one node. This may introduce implicit dependencies between the actions since each allocation adds constraints to the constraint problem of the platform. There can for example be a shared resource that both actions use. Figure 5 shows the constraint network of platform P_1 after it has been allocated nodes N_2 and N_4 from the example TST. In this example the position of the platform is implicitly shared since the first action will change the location of the platform.

A *complete allocation* is an allocation which allocates every node in a TST to a platform. A completely allocated TST defines a constraint problem that represents all the constraints for this particular allocation of the TST. As the constraints are distributed among the platforms it is in effect a distributed constraint problem. If the constraint problem is consistent then a *valid allocation* has been found and each solution can be seen as a potential execution schedule of the TST. The consistency of an allocation can be checked by a distributed constraint satisfaction problem (DCSP) solver such as the Asynchronous Weak Commitment Search (AWCS)

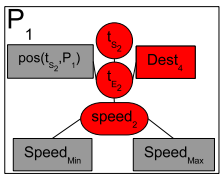

Figure 3: The parameterized platform constraints for the scan action. The red/dark variables are connected to the TST as part of the node interface.

algorithm [17] or ADOPT [13].

4. MULTI-ROBOT TASK ALLOCATION

Multi-robot task allocation (MRTA) is an important problem in the multiagent community [7, 8, 12, 16, 18, 19]. It deals with the complexities involved in taking a description of a set of tasks and deciding which of the available robots should do what. Often the problem also involves maximizing some utility function or minimizing a cost function. Important aspects of the problem are what types of tasks and robots can be described, what type of optimization is being done, and how computationally expensive the allocation is. In this section we discuss the MRTA problem and how it relates to allocating complex tasks specified as TSTs. In the process, we extend the classification introduced by Gerkey and Matarić [7, 9] with four new dimensions.

The task allocation problem can be traced back to the Optimal Assignment Problem (OAP) [6]. In OAP, m workers should be assigned to n jobs, one worker per job, where the worker-job combinations have different utilities depending on how well suited the worker is for the job. The problem is to find the optimal allocation.

The following assumptions are made in OAP: A worker can only have one job at a time. A job only needs one worker. The assignment is instantaneous. There are no more jobs to take care of later. The jobs are atomic in the sense that they do not relate to each other. Both utilities and jobs are independent. Since assigning a worker to a job does not change the utilities of other workers, the jobs can be assigned in any order. One can see that the problem has three dimensions: worker capacity, job complexity, and allocation horizon.

4.1 Classifying Multi Robot Task Allocation

The multi-robot task allocation problem is in its simplest form equal to the OAP. By varying the problem along the three OAP dimensions Gerkey and Matarić define seven more complex variants [9]. Single task robots (ST) vs. multi-task robots (MT), i.e. can a robot execute one or many tasks at the same time (worker capacity). Single robot tasks (SR) vs. multi-robot tasks (MR), where SR means that each task can be executed by a single robot, while with MR a task may need more than one robot (job complexity). The final dimension, allocation horizon, is instantaneous assignment (IA) vs. time-extended assignment (TE). In IA there is no information available to reason about further allocations, instead an allocation can be done directly with the information that is available. For TE there is more information such as information about all tasks that need to be assigned or a model of how tasks are expected to arrive in time.

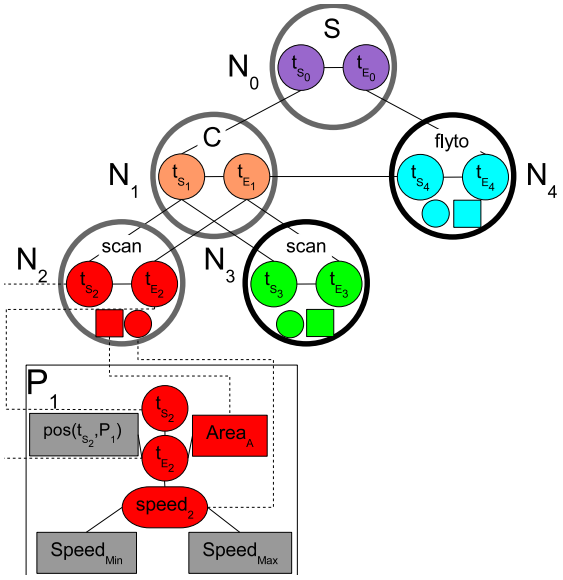

Figure 4: The combined constraint problem after allocating node N_2 to platform $\mathbf{P}_1$.

In his thesis [7], Gerkey points out that the classification does not really apply to tasks that have interrelated utilities (e.g., the utility of task 1 for platform A is dependent on whether it is also allocated task 2) and tasks that have constraints between them (e.g., a TST with sequential tasks). To cover these cases we extend the classification model with the dimensions unrelated utilities (UU) vs. interrelated utilities (IU) and independent tasks (IT) vs. constrained tasks (CT).

Another aspect of the task allocation problem is *who* is making the task allocation. In OAP, solving the allocation problem is separate from executing the allocated tasks. The allocation itself is not seen as something that has to be done by a worker, instead it is an external process. If the allocation is done by a worker, then both the tasks and the task allocation are tasks for the multiagent system. Making task allocation a task is part of the delegation concept. A delegation is a task allocation performed by a particular platform. We call this new dimension external allocation view (EV) vs. internal allocation view (IV). Whether IV is harder than EV depends on how much information about the task allocation problem that the allocator has. In EV it is assumed that all the information can be given to the external allocator. This does not have to be the case for IV.

Related to, but not directly included in the task allocation problem is the task allocation environment dimension. A task allocation environment can be even more challenging than TE, if the task allocator not only has to take into account future tasks to allocate, but also that the task allocation problem can change unexpectedly. Changes could include addition or removal of robots, changes to constraints, and changes to variables. Such environments introduce the additional problem of task re-allocation. We call this extra dimension static allocation environment (SA) vs. dynamic allocation environment (DA).

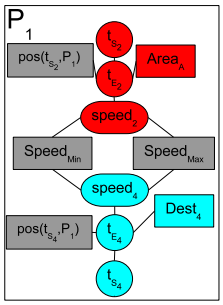

Figure 5: The parameter constraints of platform P_1 when allocated node N_2 and N_4.

4.2 Classifying Allocating TST Specified Tasks

Following the above classification, the problem of allocating a complex task according to a TST is classified as a MT-SR-TE-IU-CT-IV-DA problem. Each platform can do more than one task at a time (MT) since it is only restricted by its resources. Only one platform is needed (SR) for each task if we view each node in the TST as an individual task. If we view the entire TST as a task, then it is in the MR class. This shows that specifying a multi-robot task as a TST avoids the problem of allocating multiple robots to the same task. More generally, the class SR-CT includes parts of the MR class. Since a TST models the tasks that should be allocated and how they relate to each other the problem is in TE. Since a TST can specify constraints such as execution order and global timing, the problem is in CT. The problem is also in IU due to shared resources for example. Since allocating tasks is an active part in the delegation, the problem is in IV. In addition, the problem is in DA, meaning that we also have to think about task re-allocation.

5. AN ALGORITHM FOR ALLOCATING COMPLEX TASKS SPECIFIED BY TSTS

This section presents a heuristic search algorithm for allocating a fully expanded TST to a set of platforms. A successful allocation allocates each node to a platform and assigns values to parameters such that each task can be carried out by its assigned platform and all the constraints of the TST are satisfied. During the allocation, variables will be instantiated resulting in a schedule for executing the TST.

The algorithm starts with an empty allocation and extends it one node at a time in a depth-first order over the TST. To extend the allocation, the algorithm takes the current allocation, finds a consistent allocation of the next node, and then recursively allocates the rest of the TST. Since a partial allocation corresponds to a distributed constraint satisfaction problem, a DCSP solver is used to check whether the constraints are consistent. If all possible allocations of the next node violate the constraints, then the algorithm uses backtracking with backjumping to find the next allocation.

The algorithm is both sound and complete. It is sound since the consistency of the corresponding constraint problem is verified in each step and it is complete since every possible allocation is eventually tested. Since the algorithm is recursive the search can be distributed among multiple platforms.

To improve the search, a heuristic function is used to determine the order platforms are tested. The heuristic function is constructed by auctioning out the node to all platforms with the required capabilities. The bid is the marginal cost for the platform to accept the task relative to the current partial allocation. The cost could for example be the total time required to execute all tasks allocated to the platform.

To increase the efficiency of the backtracking, the algorithm uses backjumping to find the latest partial allocation which has a consistent allocation of the current node. This preserves the soundness as only partial allocations that are guaranteed to violate the constraints are skipped.

The AllocateTST algorithm takes a TST rooted in the node N as input and finds a valid allocation of the TST if possible. To check whether a node N can be allocated to a specific platform P the TryAllocateTST algorithm is used. It tries to allocate the top node N to P and then recursively finding an allocation of the sub-TSTs.

AllocateTST*(Node N)*

1. Find the set of candidates C for N.
2. Run an auction for N among the candidates in C and order C according to the bids.
3. For each candidate c in the ordered set C:
   (a) If TryAllocateTST(c, N) then return success.
4. Return failure.

TryAllocateTST*(Platform P, Node N)*

1. AllocateTST P to N.
2. If the allocation is inconsistent then undo the allocation and return false.
3. For each sub-TST n of N do
   (a) If AllocateTST(n) fails then undo the allocation and do a backjump.
4. An allocation has been found, return true.

The implementation of TryAllocateTST is based on the contract-net protocol [15]. For a platform A to try to allocate a TST rooted in N to platform B it sends a *call-for-proposal* (cfp) message containing the TST to platform B. If TryAllocateTST is successful then A will send a *propose* message back to A otherwise it will send a *refuse* message.

5.1 Node Auctions

Broadcasting for candidates for a node N only returns platforms with the required capabilities for the node. There is no information about the usefulness or cost of allocating the node to the candidate. Blindly testing candidates for a node is an obvious source of inefficiency. Instead, the node is auctioned out to the candidates. Each bidding platform bids its marginal cost for executing the node. I.e., taking into account all previous tasks the platform has been allocated, how much more would it cost the platform to take on the extra task. The cost could for example be the total time needed to complete all tasks. To be efficient, it is important that the cost can be computed by the platform locally. We

are currently only evaluating the cost of the current node, not the sub-TST rooted in the node. This leaves room for interesting extensions. Low bids are favorable and the candidates are sorted according to their bids. The bids are used as a heuristic function that increases the chance of finding a suitable platform early in the search.

5.2 Distributed Backjumping

A dead-end is reached when a platform is trying to allocate a node N_k but there is no consistent allocation. The platform must then undo previous allocations until a partial allocation is found where N_k can be allocated. This is the backjump point where the backtracking will start.

More formally, the current partial allocation can be seen as the assignment $A_1, \ldots, A_k$ of platforms to each node in the sequence $N_1, \ldots, N_k$. Instead of backtracking over the next allocation for $N_1, \ldots, N_{k-1}$ as in normal chronological backtracking, the algorithm finds the node N_j with the highest index j such that a consistent allocation for N_k can be found given the partial allocation $A_1, \ldots, A_j$. The node N_j is called the *backjump point*. Using the fact that N_k must be allocated we can skip all partial allocations of $N_{j+1}, \ldots, N_{k-1}$ that do not lead to a consistent allocation of N_k.

The backjump point is found by disconnecting parts of the DCSP network and then trying all possible allocations for N_k. When the node can be allocated with parts of the network disconnected, it means that the backjump point resides in the disconnected part of the network. The localization of the backjump point continues in the previously disconnected network by recursively dividing it into smaller parts. Each new partial allocation is checked by trying to extend it with an allocation of N_k. Since the task allocation process is distributed the backjump process must also be distributed.

To describe the algorithm, the following definitions are used. A platform is *in charge* of all nodes below a node it has been allocated. The node that could not be allocated is called the *failure point*. The platform trying to find an allocation for the failure point is called *failure point allocator*. *Disconnecting* a network means temporarily removing the variables in the network from the DCSP which is equivalent to removing the corresponding allocations. When a platform disconnects networks and checks for consistency, an *activation* message is sent from the platform to the failure point allocator. The failure point allocator will then try applicable platforms for the failure point until an allocation is found or none exists. The failure point allocator sends an *allocation succeed* if an allocation is found, otherwise an *allocation failed* message.

The procedures *Search Upwards* and *Search Downwards* are used to find the backjump point, beginning with the *Search Upwards* procedure. Two different search procedures are necessary since we first have to find which platform is in control over the backjump point, and second to find the actual backjump point.

Search Upwards

1. Disconnect all child branches (that have been allocated) except the branch that contains the failure point. Signal the failure point allocator to start finding an allocation for the failure point.
   (a) If the failed node can be allocated, reconnect all child branches and start searching for the backjump point by calling Search Downwards.
   (b) If no allocation can be found, then do a Search Upwards starting from the parent of the node. If the node has no parent then there is no allocation.

Search Downwards

1. Disconnect child branches one at the time in the reverse order they were allocated and check the consistency. If the network is consistent then the backjump point is in that branch.
2. When a branch containing the backjump point is located, check if the child branch has a composite action node as the top-node. In that case, do a recursive Search Downwards starting at that node. Otherwise, the backjump point has been found.

6. A UAV CASE STUDY

On December 26, 2004, a devastating earthquake of high magnitude occurred off the west coast of Sumatra. This resulted in a tsunami which hit the coasts of India, Sri Lanka, Thailand, Indonesia, and many other islands. Both the earthquake and the tsunami caused great devastation. During the initial stages of the catastrophe, there was a great deal of confusion and chaos in setting into motion rescue operations in such wide geographic areas. The problem was exacerbated by a shortage of manpower, supplies, and machinery. The highest priorities in the initial stages of the disaster were searching for survivors in many isolated areas where road systems had become inaccessible and providing relief in the form of delivery of food, water, and medical supplies.

Let us assume that one has access to a fleet of autonomous unmanned helicopter systems with ground operation facilities. How could such a resource be used in the real-life scenario described?

A prerequisite for the successful operation would be the existence of a multiagent (UAV platforms, ground operators, etc.) software infrastructure for assisting emergency services. At the very least, one would require the system to allow mixed-initiative interaction with multiple platforms and ground operators in a robust, safe, and dependable manner. As far as the individual platforms are concerned, one would require a number of different capabilities, not necessarily shared by each individual platform, but by the fleet in total. These capabilities would include: the ability to scan and search for salient entities such as injured humans, building structures, or vehicles; the ability to monitor or survey these salient points of interest and continually collect and communicate information back to ground operators and other platforms to keep them situationally aware of current conditions; and the ability to deliver supplies or resources to these salient points of interest if required. For example, identified injured persons should immediately receive a relief package containing food, water, and medical supplies.

To be more specific in terms of the scenario, we can assume there are two separate legs or parts to the emergency relief scenario in the context sketched previously.

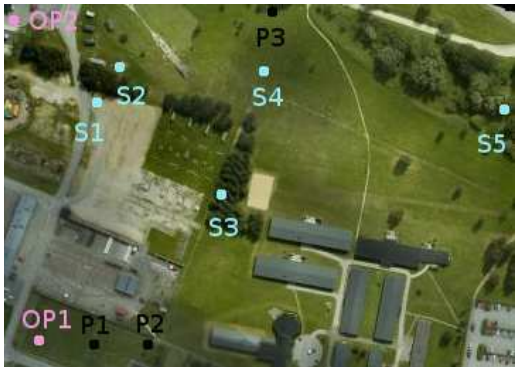

Figure 6: The disaster area with platforms P_1–P_3, survivors S_1–S_5, and operators OP_1 and OP_2.

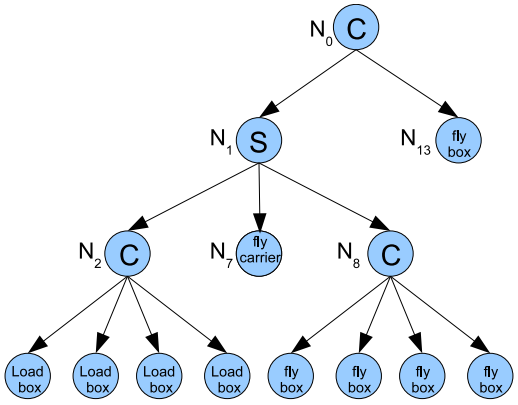

Figure 7: The TST for the logistics scenario.

Leg I In the first part of the scenario, it is essential that for specific geographic areas, the UAV platforms should cooperatively scan large regions in an attempt to identify injured persons. The result of such a cooperative scan would be a saliency map pinpointing potential victims and their geographical coordinates and associating sensory output such as high resolution photos and thermal images with the potential victims. The saliency map could then be used directly by emergency services or passed on to other UAVs as a basis for additional tasks.

Leg II In the second part of the scenario, the saliency map from Leg I would be used for generating and executing a plan for the UAVs to deliver relief packages to the injured. This should also be done in a cooperative manner.

In this paper, we will focus on the second leg, which is an example of a logistics scenario. One approach to the first leg is described in [14].

One approach to solving logistics problems is to use a task planner to generate a sequence of actions that will transport each box to its destination. Each action must then be executed by a UAV. We have previously shown how to generate pre-allocated plans and monitor their execution [3, 11]. In this paper we show how a plan without explicit allocations expressed as a TST can be allocated to a set of UAV platforms which where not known at the time of planning.

6.1 The TST for the Logistics Scenario

In this particular scenario, shown in Figure 6, five survivors (S_1–S_5) are found in Leg I, and there are three platforms (P_1–P_3) and one carrier available. To start Leg II, the operator creates a TST, for example using a planner, that will achieve the goal of distributing relief packages to all survivor locations in the saliency map [11]. The resulting TST is shown in Figure 7. The TST contains a sub-TST (N_1–N_{12}) for loading a carrier with four boxes (N_2–N_6), delivering the carrier (N_7), and unloading the packages from the carrier and delivering them to the survivors (N_8–N_{12}). A package must also be delivered to the survivor in the right uppermost part of the region, far away from where most of the survivors were found (N_{13}). The delivery of packages can be done concurrently to save time, but the loading, moving, and unloading of the carrier is a sequential operation. UAVs and equipment should be allocated carefully to assure that all relief packages reach their destinations in time.

Another operator OP_2 is performing a scan mission, with the platforms P_3 and P_4 north of the area in Figure 6. P_3 is currently idle and OP_1 is therefore allowed to borrow it.

6.2 Allocating the Logistics Scenario Tasks

To allocate the TST in Figure 7 the operator OP_1 invokes AllocateTST on the top node N_0. After an auction between P_1 and P_2 for N_0, OP_1 sends a *cfp* message with the TST to the winner P_1. This invokes TryAllocateTST for N_0 on P_1.

P_1 is now responsible for N_0 and for allocating the remaining nodes in the TST. The task allocation algorithm traverses the TST in depth-first order, so P_1 should first find a platform for node N_1, and when the entire sub-TST rooted in N_1 is allocated, find an allocation for node N_{13}. Node N_1 and N_2 are two composite action nodes which have the same marginal cost for all platforms. P_1 therefore recursively allocates N_1 and N_2 to itself. The constraints from nodes N_0–N_2 are added to the constraint network of P_1. The network is consistent because the composite action nodes describe a schedule without any restrictions.

Below node N_2 are four primitive action nodes. Since P_1 is responsible for N_2, it tries to allocate them one at the time. For primitive action nodes, the choice of platform is the key to a successful allocation. This is because of each platform's unique state, constraint model for the action, and available resources. The candidates for node N_3 are platform P_1 and P_2. P_1 is closest to the package depot, and therefore gives the best bid for the node. P_1 is allocated to N_3. For node N_4, platform P_1 is still the best choice, and it is allocated to N_4. Given the new position of P_1 after being allocated N_3 and N_4, P_2 is now closest to the depot resulting in the lowest bid and is allocated to N_5 and N_6. The schedule defined by nodes N_0–N_2 is now constrained further by how long it takes for P_1 and P_2 to carry out action nodes N_3–N_6. The constraint network is now shared between platforms P_1 and P_2.

The next node to allocate for P_1 is node N_7, the carrier delivery node. P_1 is the only platform that has the fly carrier capability and is allocated the node. Continuing with the nodes N_8–N_{12}, the platform with the lowest bid for each node is platform P_1, since it is in the area after delivering the carrier. P_1, is therefore allocated to nodes N_8–N_{12}.

The final node, N_{13}, is allocated to platform P_2 and the allocation is complete. The only non-local information used by P_1 was the capabilities of the available platforms which was gathered through a broadcast. Everything else is local. The bids are made by each platform based on local information and the consistency of the constraint network is checked through distributed constraint satisfaction techniques.

The total mission time is 58 minutes, much longer than the operator expected. Since the constraint problem defined by the allocation to the TST is shared between the platforms, it is possible for the operator to modify the constraint problem by adding more constraints, and in this way modify the task allocation. The operator puts a time constraint on the mission, restricting the total time to 30 minutes.

To re-allocate the modified TST, operator OP_1 sends a reject-proposal to platform P_1. The added time constraint to the mission makes the current allocation inconsistent. The last allocated node must therefore be re-allocated. However, no platform for N_{13} can make the allocation consistent, not even the newly added P_3. Backtracking starts. Platform P_1 is in charge, since it is responsible for allocating node N_{13}. The N_1 sub-network is disconnected. Trying different platforms for the node N_{13}, P_1 discovers that N_{13} can be allocated to P_2. P_1 sends a backjump-search message to the platform in charge of the sub-TST with top-node N_1, which happens to be P_1, to start an Upward Search. When receiving the message P_1 continues the search for the backjump point. Since removing all constraints due to the allocation of node N_1 and its children made the problem consistent, the backjump point is in the sub-TST rooted in N_1. Removing the allocations for sub-tree N_8 does not make the problem consistent so further backjumping is necessary. Notice that with a single consistency check the algorithm could show that no possible allocation of N_8 and its children can lead to a consistent allocation of N_{13}. Removing the allocation for node N_7 does not make a difference either. Removing the allocations for sub-TST N_2 makes the problem consistent. When finding an allocation of N_{13} after removing the constraints from N_6 the allocation process continues from N_6 and tries the next platform for the node, P_1.

When the allocation reaches node N_{11} it is discovered that since P_1 has taken on nodes N_3–N_8, there is not enough time left for P_1 to unload the last two packages from the carrier. Instead P_3, even though it made a higher bid for N_{11}–N_{12}, is allocated to both nodes. Finally platform P_2 is allocated to node N_{13}. It turns out that since platform P_2 helped P_1 loading the carrier, it has not enough time to deliver the final package. Instead, a new backjump point search starts, finding node N_5, and continuing from there. This time around, nodes N_3–N_9 are allocated to platform P_1, platform P_3 is allocated to node N_{10}–N_{12}, and platform P_2 is allocated to node N_{13}. The allocation is consistent. The allocation algorithm finishes on platform P_1, by sending a propose message back to the operator. The operator inspects the allocation and approves it, thereby starting the execution of the mission.

Scenario	A	D	C	kB	Makespan (range)
Log5-2P-A	12	16	6	52	253
Log5-2P-R	-	13	6	27	289 (257–321)
Log5-4P-A	31	18	7	89	224
Log5-4P-R	-	20	6	142	221 (201–242)
Log17-2P-A	40	46	32	177	543
Log17-2P-R	-	49	32	114	561 (451–679)
Log17-4P-A	141	78	18	160	365
Log17-4P-R	-	73	24	142	389 (311–511)

Table 1: Evaluation of the task allocation algorithm.

6.3 Preliminary Empirical Evaluation

To get an initial feeling for the runtime cost of the algorithm and the benefit of using node auctions as a heuristic function we have tried a number of different settings. We used both a TST like the one in the scenario with 5 boxes (14 nodes) and a larger TST with 17 boxes (50 nodes). The algorithm was tried on each scenario with both 2 or 4 platforms and with or without node auctions. The scenarios which did not use node auction used a random order of the nodes. To get a better estimation of the randomized test we ran them 10 times and took the average results.

The results are shown in Table 1. To measure the cost and quality of the algorithm we measured the number of messages related to auctions (A), delegation (D), and constraint satisfaction (C), the total size of the messages (kB), and the total makespan of the resulting allocation. The makespan can be seen as a quality measure of the allocation. The shorter it is the more efficient the allocation is.

From these preliminary results we can see that node auctions use more messages and result in a slightly higher quality. The benefit is expected to increase as the TST complexity grows and a more careful choice of allocation is required. In the example TSTs very little backtracking was necessary.

7. RELATED WORK

The closest work to allocating TSTs is the work on task allocation for task trees [16, 18, 19]. In task trees, tasks are related to each other either by precedence constraints or by compositions as expressed by logical connectives. The authors call this "complex task allocation". A major difference is that these task trees can not express interrelated utilities (IU), which TSTs can.

Many task allocation algorithms are auction-based [2, 18]. There, tasks are auctioned out and allocated to the agent that makes the best bid. Bids are determined by a utility function. The auction concept decentralizes the task allocation process which is very useful especially in multi-robot systems, where centralized solutions are impractical. For tasks that have unrelated utilities, this approach has been very successful. The reason is that UU guarantees that each task can be treated as an independent entity, and can be auctioned out without affecting other parts of the alloca-

tion. This means that a robot does not have to take other tasks into consideration when making a bid.

In complex task allocation sub-tasks may not be independent. A complex task has structure and there are relations between its atomic tasks. It is also often the case that a complex task must be allocated to a group of agents, creating relations between the agents relative to the task. Complex task allocation must therefore take into account synergy effects between allocations which influence the bids for tasks. A bid could for example be different depending on other commitments of the platform.

More advanced auction protocols have been developed to handle dependencies among tasks. These are constructed to deal with complementarities (substitution effects, which we call interrelated utilities). Examples are sequential single item auctions [10] and combinatorial auctions [1]. These auctions typically handle that different combinations of tasks have different bids, which can be compared to our model where different sets of allocations result in different restrictions to the constraint network between the platforms.

The sequential single item (SSI) auction [10] is of special interest as it is similar to our algorithm. In SSI auctions, the tasks are auctioned out in sequence, one at a time to make sure the new task fits with the previous allocations. Normally SSI auctions are applied to problems where it is easy to find a solution but it is hard to find a good solution. They are therefore normally not complete for problems where it is hard to find a solution, like with TST allocation.

Combinatorial auctions deal with complementarities by bidding on bundles containing multiple items. Each bidder places bids on all the bundles that are of interest, which could be exponentially many. The auctioneer must then select the best set of bids, called the winner determination problem, which is NP-hard [1]. This means that even in the best case there is a very high computational cost involved in using combinatorial auctions.

8. CONCLUSIONS

Two central problems in our research with collaborative unmanned aerial vehicles are to define complex mixed-initiative missions and given a mission find UAV platforms that can execute it. We have previously introduced a formal delegation framework and within that proposed task specification trees as a highly expressive specification language for multiagent tasks that supports mixed-initiative delegation with adjustable autonomy. In this paper we have discussed the problem of allocating complex tasks to robots. We extended the multi-robot task allocation classification introduced by Gerkey and Matarić [9] with four new dimensions and argued that allocating task specification trees is more challenging than most allocation problems currently considered. The problem of allocating TSTs to robot platforms was defined and a heuristic algorithm for finding a consistent allocation was presented. The algorithm recursively searches among the potential allocations in a distributed manner and uses distributed constraint satisfaction techniques to check if an allocation satisfies the constraints. We also presented a detailed case study with a team of unmanned aerial vehicles assisting in a challenging emergency services scenario.

In conclusion, specifying and allocating complex tasks are important research problems in multiagent systems, especially when dealing with robotic agents in the real world. The presented approach takes another step towards a practical collaborative multi-robot system.

9. REFERENCES

[1] S. de Vries and R. Vohra. Combinatorial auctions: A survey. *Journal on Computing*, 15(3):284–309, 2003.

[2] M. Dias, R. Zlot, N. Kalra, and A. Stentz. Market-based multirobot coordination: a survey and analysis. *Proc. of IEEE*, 94(1):1257 – 1270, 2006.

[3] P. Doherty, J. Kvarnström, and F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 2009.

[4] P. Doherty, D. Landén, and F. Heintz. A distributed task specification language for mixed-initiative delegation. In *Proc. PRIMA*, 2010.

[5] P. Doherty and C. J.-J. Meyer. Towards a delegation framework for aerial robotic mission scenarios. In *Proc. CIA*, 2007.

[6] D. Gale. *The Theory of Linear Economic Models*. McGraw-Hill Book Company, Inc., 1960.

[7] B. Gerkey. *On multi-robot task allocation*. PhD thesis, 2003.

[8] B. Gerkey and M. Mataric. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation*, 2001.

[9] B. Gerkey and M. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *Int. Journal of Robotic Research*, 23(9):939–954, 2004.

[10] S. Koenig, P. Keskinocak, and C. Tovey. Progress on agent coordination with cooperative auctions. In *Proc. AAAI*, 2010.

[11] J. Kvarnström and P. Doherty. Automated planning for collaborative uav systems. In *Proc. ICARCV*, 2010.

[12] T. Lemaire, R. Alami, and S. Lacroix. A distributed tasks allocation scheme in multi-uav context. In *Proc. ICRA*, 2004.

[13] P. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *AI*, 161, 2006.

[14] P. Rudol and P. Doherty. Human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. In *Proc. of the IEEE Aerospace Conference*, 2008.

[15] R. Smith. The contract net protocol. *IEEE Transactions on Computers*, C-29(12), 1980.

[16] A. Viguria, I. Maza, and A. Ollero. Distributed service-based cooperation in aerial/ground robot teams applied to fire detection and extinguishing missions. *Advanced Robotics*, 24:1–23, 2010.

[17] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. CP*, 1995.

[18] R. Zlot. *An auction-based approach to complex task allocation for multirobot teams*. PhD thesis, 2006.

[19] R. Zlot and A. Stentz. Complex task allocation for multiple robots. In *Proc. ICRA*, 2005.