Unmanned Systems, Vol. 0, No. 0 (2013) 1–39

High-level Mission Specification and Planning for Collaborative Unmanned Aircraft Systems using Delegation

Patrick Doherty[a], Fredrik Heintz[a] and Jonas Kvarnström[a]

[a]*Linköping University, S-581 83 Linköping, Sweden*
E-mail: patrick.doherty@liu.se, fredrik.heintz@liu.se, jonas.kvarnstrom@liu.se

Automated specification, generation and execution of high level missions involving one or more heterogeneous unmanned aircraft systems is in its infancy. Much previous effort has been focused on the development of air vehicle platforms themselves together with the avionics and sensor subsystems that implement basic navigational skills. In order to increase the degree of autonomy in such systems so they can successfully participate in more complex mission scenarios such as those considered in emergency rescue that also include ongoing interactions with human operators, new architectural components and functionalities will be required to aid not only human operators in mission planning, but also the unmanned aircraft systems themselves in the automatic generation, execution and partial verification of mission plans to achieve mission goals. This article proposes a formal framework and architecture based on the unifying concept of delegation that can be used for the automated specification, generation and execution of high-level collaborative missions involving one or more air vehicles platforms and human operators. We describe an agent-based software architecture, a temporal logic based mission specification language, a distributed temporal planner and a task specification language that when integrated provide a basis for the generation, instantiation and execution of complex collaborative missions on heterogeneous air vehicle systems. A prototype of the framework is operational in a number of autonomous unmanned aircraft systems developed in our research lab.

1. Introduction

Much of the recent research activity with Unmanned Aircraft Systems (UASs) has focused primarily on the air vehicle (AV) itself, together with the avionics and sensor sub-systems. Primary focus has been placed on the navigation subsystem together with low-level control combined with motion planners that allow a UAS to operate with limited autonomy. The control kernel implements diverse control modes such as take-off, landing, flying to waypoints and hovering (in the case of rotor-based systems). Sensor payloads are then used to gather data after positioning the AV at salient points of interest.

Development of this type of low-level autonomy has been impressive, resulting in many AV systems that with the help of human operators can autonomously execute missions of moderate complexity. Specification of such missions is often based on the manual or semi-manual construction of a waypoint database, where waypoints may be annotated with sensor tasks to be achieved at each of these points. Such a means of specifying missions is often time consuming and also prone to error due to the low level of abstraction used and to the lack of automation in generating such plans in addition to the lack of automatic verification of the correctness of the mission.

Although these capabilities provide the basic functionality for autonomous AVs, if one is interested in increasing the complexity of the missions executed and the usefulness of UASs, much more is required. This ”much more” is very much what this article will focus on. The collection of functionalities and capabilities required to automate both the process of specifying and generating complex missions, instantiating their execution in the AV, monitoring the execution and repairing mission plans when things go wrong, commonly goes under the umbrella term ”high autonomy”. Systems with high autonomy require additional architectural support beyond what one commonly uses to support the low-level autonomy in such systems. Furthermore, one has to ensure that each of the architectural components that support both low and high autonomy are fully integrated in the resulting system.

There has also been a recent trend in research in the direction of collaborative systems where rather than focusing on the use of a single AV to complete a mission, several heterogeneous UASs are required to participate in a mission together with one or more human operators with continual interaction between them during the achievement of mission goals. This added dimension increases the complexity of the functionalities required in the individual AVs in addition to requiring additional architectural support for collaborative UAS missions. Tools for formally verifying such missions are even more important in this context due to the increased complexity of the missions.

In this article we will propose a conceptual framework and architecture to support the specification, generation and execution of collaborative missions involving heterogeneous UASs and human operators. The main components that will be described are:

- A high-level mission specification language based on the use of Temporal Action Logic (TAL) to formally specify, reason about and verify both single- and multi-platform mission specifications.
- Task Specification Trees (TSTs), a pragmatically motivated, distributable and extensible structure for defining single- and multi-platform tasks and missions. TSTs are executable on robotic systems that support their executability. A formal semantics of TSTs is provided by TAL.
- An automated planner called TFPOP, capable of generating both single- and multi-platform mission specifications in the high-level mission specification language that are translatable into executable TSTs.
- A generic multi-agent software architecture based on the concept of delegation that can be added to and integrated with existing legacy UAS architectures to support collaborative missions. Delegation is formalized as a speech act. The architecture integrates the use of the high-level mission specification language, the delegation processes, TSTs, and the automated planner.

Most importantly, the conceptual framework and architecture have been implemented in prototype and integrated with a number of research Unmanned Aerial Vehicle (UAV) platforms used in our group and described below.

1.1. *Research UAV Platforms*

For well over a decade, the UASTech Lab[a] at Linköping University has been involved in the development of autonomous unmanned aerial vehicles (UAVs). The two platforms described below have been integrated with the functionalities and architectures described in this article.

The UASTech RMAX UAV platform [1–4] is a slightly modified Yamaha RMAX helicopter (Figure 1). It has a total length of 3.6 meters (including main rotor) and is powered by a 21 hp two-stroke engine with a maximum takeoff weight of 95 kg. The on-board system contains three PC104 embedded computers. The primary flight control (PFC) system includes a Pentium III at 700 MHz, a wireless Ethernet bridge, a GPS receiver, and several additional sensors including a barometric altitude sensor. The PFC is connected to the RMAX helicopter through the Yamaha Attitude Sensor (YAS) and Yamaha Attitude Control System (YACS), as well as to an image processing computer and a computer responsible for deliberative capabilities. The deliberative/reactive system (DRC) runs on the second PC104 embedded computer (Pentium-M 1.4 GHz) and executes all high-end autonomous functionalities such as mission or path planning. Network communication between computers is physically realized with serial lines RS232C and Ethernet. The image processing system (IPC) runs on the third PC104 embedded Pentium III 700 MHz computer.

The camera platform suspended under the UAV fuselage is vibration isolated by a system of springs. The platform consists of a Sony FCB-780P CCD block camera and a Thermal-Eye 3600AS miniature infrared camera mounted rigidly on a Pan-Tilt Unit (PTU) as presented in Figure 2. The video footage from both cameras is recorded at a full frame rate by two MiniDV recorders to allow processing after a flight.

Figure 1. Yamaha RMAX.

Figure 2. On-board camera system mounted on a pan-tilt unit.

The UASTech LinkQuad MAV platform[b] is the newest addition to our UAV fleet. It is a highly versatile autonomous quadrotor Micro Aerial Vehicle. The platform's airframe is characterized by a modular design which allows for easy reconfiguration to adopt to a variety of applications. Due to its compact design (below 70 centimeters tip-to-tip) the platform is suitable for both indoor and outdoor use. It is equipped with custom designed optimized propellers which contribute to an endurance of up to 40 minutes. Depending on the required flight time, one or two 2.7 Ah batteries can be placed inside an easily swappable battery module. The maximum take-off weight of the LinkQuad is 1.4 kilograms with up to 300 grams of payload.

[a] www.ida.liu.se/divisions/aiics/

[b] www.uastech.se

The LinkQuad is equipped with an in-house designed flight control board – the LinkBoard. The LinkBoard has a modular design that allows for adjusting the available computational power depending on mission requirements. Due to the available onboard computational power, it has been used for computationally demanding applications such as the implementation of an autonomous indoor vision-based navigation system with all computation performed on-board [5,6]. In the full configuration, the LinkBoard weighs 30 grams, has very low power consumption and has a footprint smaller than a credit card. The system is based on two ARM Cortex-M3 microcontrollers running at 72 MHz which implement the core flight functionalities, and optionally, up to two Gumstix Overo boards for user software modules. The LinkBoard includes a three-axis accelerometer, three rate gyroscopes, and absolute and differential pressure sensors for estimation of the altitude and the air speed, respectively. The LinkBoard features a number of interfaces which allow for easy extension and integration of additional equipment. It supports various external modules such as a laser range finder, analogue and digital cameras on a gimbal, a GPS receiver, and a magnetometer.

Figure 3. LinkQuad MAV Platform.

1.2. *Motivating Scenarios*

Specific target scenarios for the use of autonomous unmanned aircraft include environmental monitoring, search and rescue missions, and assisting emergency services in scenarios such as earthquakes, flooding or forest fires. For example, in November 2011, a powerful earthquake off the coast of Japan triggered a tsunami with devastating effects, including thousands of dead and injured as well as extensive damage to cities and villages through surging water. Similar natural disasters have occurred in for example Haiti and China as well as off the coast of Sumatra, also with catastrophic effects. In each of these disasters, unmanned aircraft could have been of assistance in a variety of ways.

As one of many immediate effects of the tsunami in Japan, a number of villages near the coast were completely isolated from the rest of the world for as much as twelve days when both bridges and phone lines were swept away, before debris could be cleared and temporary bridges could be built. Now suppose that an emergency response unit had a small fleet of unmanned aircraft such as the RMAXs at its disposal. Such aircraft could then have assisted by rapidly delivering medicine, food, water, or whatever other supplies were needed in an isolated village. They could also have supported emergency responders in situation assessment and other tasks such as searching for injured people in areas that are otherwise inaccessible.

Another effect, which became increasingly apparent over time, was the extensive damage to the Fukushima Daiichi nuclear plant which later resulted in a complete meltdown in three reactors. The exact level of damage was initially difficult to assess due to the danger in sending human personnel into such conditions. Here unmanned aircraft could immediately have assisted in monitoring radiation levels and transmitting video feeds from a considerably closer range, with smaller aircraft such as the LinkQuad entering buildings to assess damage.

Aspects of these complex scenarios will be used as examples in the article to show how such missions would be specified, generated and executed in the proposed framework. Successful deployment of autonomous air vehicles or other unmanned systems in scenarios of this complexity requires research and development in a variety of areas related to hardware as well as software systems.

1.3. *Mission Specification, Generation and Execution*

In this article we focus on the problem of specifying, generating and executing the collaborative missions that the air vehicles involved would be required to perform in this problem domain as well as in others. The ability to do this clearly and concisely is fundamentally important for an unmanned system to be practically useful, not least when operators are under time pressure, and requires a suitable *mission specification language* that should satisfy a variety of requirements and desires. Creating such a language is a highly non-trivial task in itself.

For example, the language should be comprehensible to humans and not only useful as an intermediate representation both generated and received by software. It should therefore provide clear, succinct language constructs that are easily understood by operators and implementors, allowing missions to be specified at a comparatively high level of abstraction. At the same time intuitions are not sufficient: A strict formal semantics must be available in order for users, system designers and implementors to be able to agree upon exactly what a given construct is intended to mean, with no room for alternative interpretations.

Additionally there should be a close connection to how missions are pragmatically executed in actual robotic systems, allowing the actual semantics of the language to be used and integrated even at the execution level and thereby facilitating the validation of the system as a whole. A principled foundation where these issues are considered, for single platforms (vehicles) as well as for fleets of homogeneous or heterogeneous platforms, is essential for these types of systems to be accepted by aviation authorities and in society, especially outside the type of emergency scenarios considered above.

Perhaps surprisingly, an essential aspect of a mission specification language is also what can be left *unspecified* or *incom-*

pletely specified. For example, one may want to state that a certain area should be scanned using a color camera at an altitude of 30 to 50 meters, and that the aircraft is then allowed to land either at two out of four predefined bases or within a geometrically specified alternate landing area. Allowing these parameters to be determined at a later time, within given restrictions, can be essential for the ability to complete the mission in changing circumstances. Thus, a mission specification language should have the ability to specify flexible *constraints* on various forms of mission parameters, which can for example be spatial or temporal in nature – an ability closely related to the concept of adjustable autonomy. Additionally, generic mission patterns often need to be constrained dynamically and incrementally relative to environmental and contingent conditions in the field. Constraints therefore offer a natural way to enhance missions.

Taking this one step further, it would be preferable if parts of a mission could be specified in terms of declarative *goals* to be achieved in addition to the actions to be performed in other parts of the mission. This combination has the potential of relieving an operator of a great deal of work involved in defining every step involved in a mission, while still permitting detailed control where desired. Support for partly goal-based specifications can also allow distinct platforms to adapt their part of a mission more closely to their own abilities. Each unmanned vehicle involved could then use general techniques such as automated planning to determine which actions could and should be used to achieve their goals, potentially even to the extent of enlisting the aid of others. The resulting actions would then be merged into the final mission specification.

Given a mission specification there must be a principled and effective means for an unmanned system to determine whether it can actually be performed, and if so, how. The most obvious reason why this can be an interesting and difficult problem is perhaps the presence of explicitly specified constraints and goals that can leave a plethora of potential solutions to be examined by the unmanned system itself. However, the problem exists even without these complicating factors. For example, a mission designer may not have full knowledge of the flight envelope of each aircraft and therefore cannot directly determine exactly how much time a certain mission would take or whether it would be feasible to meet a given deadline. Determining this can require the use of knowledge and procedures, such as motion planners, that are only known in detail to the unmanned systems themselves. For a fleet of unmanned platforms, the exact *task allocation* specifying which platform performs which actions can also be crucial to the feasibility of a complex mission. Furthermore, some platforms may already be assigned tasks from other missions, in which case potential conflicts must be taken into account.

1.4. *Article Roadmap*

We will first discuss the software architecture extensions currently used on the UASTech unmanned aircraft. In Section 2 we show how this architecture and its use of *delegation* as a unifying and primary concept lends itself to resolving several problems discussed above. A side effect of using delegation as a unifying concept is that it provides clear conceptualization of adjustable autonomy and mixed-initiative interaction. This provides a concrete setting in which mission specifications can be considered. Delegation is formally characterized as a speech act. This formal specification is used as a basis for the implementation of delegation processes in the architecture.

We then present a mission specification language that provides a formal semantics for missions specified in terms of temporal composite actions (Section 3). Elementary actions in this language are specified using Temporal Action Logic (TAL), a well-established and highly expressive logic for defining and reasoning about actions and their effects. Constructs such as sequential composition, parallel composition, conditions, and loops are introduced to permit more complex action structures, a process that can be taken to the level of complete multi-platform missions. The language is specifically designed to allow partial mission specifications with constraints, including resource requirements and temporal deadlines.

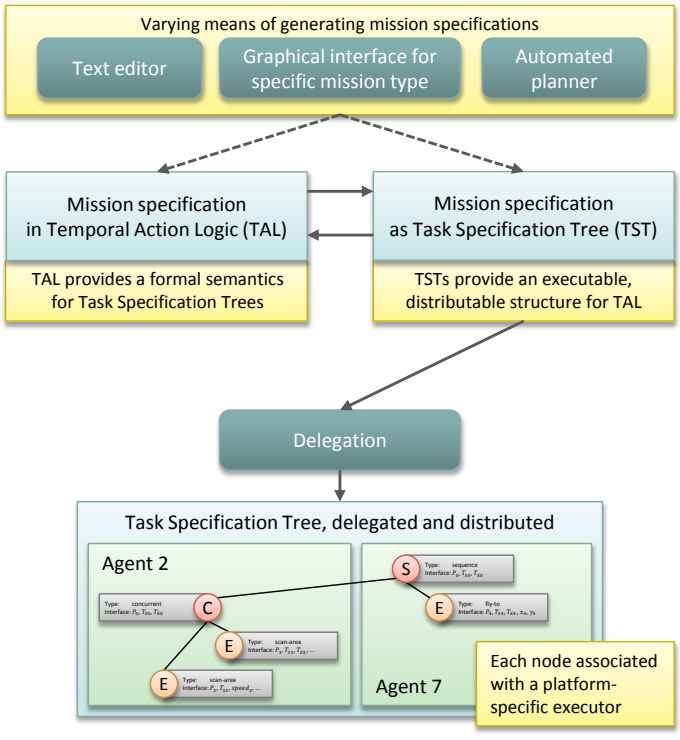

Figure 4. Mission specification, translation and delegation.

As illustrated in Figure 4, the mission specification language is shown to correspond closely to the expressivity of Task Specification Trees (TSTs), a pragmatically motivated, distributable and extensible structure for defining single-platform and multi-platform tasks and missions (Section 4). Missions can be defined in TAL or as TSTs and can be translated in either direction. Like composite actions, TSTs themselves are entirely declarative in nature. At the same time there is a close integration with execution aspects of actions and action composition constructs through the coupling of *executors* to specific types of

TST nodes. While executors must satisfy the declarative requirements of the corresponding nodes, their implementation can be platform-specific, allowing the exact mode of execution for a particular task to be platform-dependent as is often necessary when heterogeneous platforms are used.

An automated planner called TFPOP, capable of generating distributable mission specifications that are transformable into task specification trees, is presented (Section 5). We then show how the high-level concept of delegation can be realized in practice using TSTs as the concrete task representation, where the executability of a mission can be verified effectively using constraint solving techniques (Section 6). Finally, we show how delegation and planning can be interleaved and integrated in several distinct ways (Section 7).

2. A Delegation-Based Framework and Architecture

The mission specification language, task representation and planning functionalities discussed in this article are realized in the context of a concrete agent-based software architecture [7]. This architecture is currently used on board all UASTech unmanned aircraft, but the intention is for the principles and solutions developed in the project to be applicable to a wide variety of robotic systems. Aircraft are thus only one example of potential robotic *platforms*, which may also include for example ground robots.

As we are interested in autonomy and semi-autonomy, we view the combination of a platform and its associated software as an *agent*. Humans interacting with platforms through for example ground control stations and other interfaces are also considered to be agents. Taken together, these agents form a collaborative system where all participants can cooperate to perform missions. One aspect of a collaborative system is that all agents are conceptually equal and independent in the sense that there is no predefined control structure, hierarchical or otherwise. A consequence is that the control structure can be determined on a mission-to-mission basis and dynamically changed during a mission.

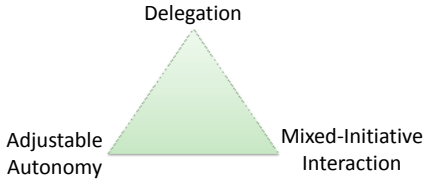

Figure 5. A conceptual triad of concepts related to collaboration.

Part of the foundation for collaboration in this architecture is a framework based on a triad of closely interdependent concepts: *delegation*, *adjustable autonomy* and *mixed-initiative interaction* (Figure 5). Among these concepts, delegation is the key [7, 8], providing a bridge that connects mixed-initiative interaction and adjustable autonomy and clarifies their intended relation in an autonomous system. Each of these concepts also has the potential of providing insights into desirable properties of mission specifications, and will therefore be further elucidated below.

Delegation. When we view an unmanned system as an agent acting on behalf of humans, it is also natural to view the assignment of a complex mission to that system as *delegation* – by definition, the act of assigning authority and responsibility to another person, or in a wider sense an agent, in order to carry out specific activities. Delegation as used in the current collaboration framework always assigns the overall responsibility for a mission to *one* distinct agent, not a group[c]. At the same time the concept is by nature recursive: To ensure that a complex mission is carried out in its entirety, an agent may have to enlist the aid of others, which can then be delegated particular parts or aspects of the mission. This results in a network of responsibilities between the agents involved and can continue down to the delegation of elementary, indivisible actions. To avoid having to name even such actions "missions", we instead use "task" as the unifying concept for everything that can be delegated to an autonomous system.

A request for an agent to take the responsibility for a task is then abstractly and concisely modeled as a form of delegation where a *delegator* delegates to a *contractor* a specific *task* to be performed under a set of *constraints*. Informally, an agent receiving a delegation request must verify that to the best of its knowledge, it will be able to perform the associated task under the given constraints, which may for example concern resource usage or temporal and spatial aspects of the mission. If this is the case, it can accept the delegation and is then committed to doing everything in its power to ensure the task is carried out. If not, it must refuse. This will be formalized in Section 2.1.

Both the task and the constraints should be represented in a unified mission specification language. As will be shown in Sections 3 and 4, the language used in our concrete realization of the delegation concept allows the use of complex tasks that can also involve goals to be achieved. If all or part of a task consists of a goal, a task planner is used to generate a suitable plan that achieves the goal (Sections 5 and 7).

Adjustable Autonomy. An autonomous unmanned system should support a wide spectrum of autonomy levels, where some missions require the system to be controlled "from above" at a very detailed level while others permit it to determine by itself how a given goal should be achieved. The system should then allow the degree of autonomy actually used to achieve a particular task at any given time to vary within the limitations mandated by the delegator, which may be a human operator or another autonomous system. If the task cannot be performed within the given limitations, the permitted level of autonomy cannot be increased unless a specific agreement is made.

One can develop a principled means of adjusting autonomy through the use of the *task* and *constraints*. A task delegated with only a goal and no plan, with few constraints, allows the robot to use much of its potential for autonomy in solving the task. The exact level of autonomy used could then be varied during execution depending on the current situation. On the other hand, a task

[c]This restriction can be lifted to allow for team to team delegation, but this extension will not be considered in this article.

specified as a sequence of actions and many constraints allows only very limited autonomy. It may even be the case that the delegator does not allow the contractor to recursively delegate.

Mixed-Initiative Interaction. By mixed-initiative interaction, we mean that negotiation between a human and a robotic system such as an unmanned aircraft can be initiated by either agent. This allows a mission to take advantage of each of their skills, capacities and knowledge in developing and elaborating a mission specification, executing the specification, and adapting to contingencies during execution.

Mixed-initiative interaction involves a broad set of theoretical and pragmatic issues. One central aspect is the ability of a ground operator (GOP) to delegate tasks to an unmanned system (US), and symmetrically, the ability of a US to delegate tasks to a GOP. A system can then adjust its level of autonomy dynamically during mission execution and can in particular choose to delegate certain decision making tasks to others if this is expected to be beneficial. Issues pertaining to safety, security, trust, etc., have to be dealt with in this interaction process and can be formalized as particular types of constraints.

2.1. *A Formal Specification of Delegation Using Speech Acts*

The concept of delegation requires a formal specification and semantics amenable to pragmatic grounding and implementation in a software system.

As a starting point, Castelfranchi and Falcone [9, 10] provide an informal discussion about delegation as a social concept building on a BDI model, where agents have beliefs, goals, intentions, and plans [11]. This discussion is illuminating but their specification lacks a formal semantics for the operators used.

Based on intuitions from this work, we have earlier provided a formal characterization of the concept of *strong* delegation: The form that is appropriate in delegating aspects of a mission, where the delegation is explicitly performed, there is mutual awareness of the delegation, and part of the result is a social commitment between the agents involved. Speech acts [12, 13] are chosen as a means of formally specifying the concept of delegation. The characterization is built on the definition and use of a newly defined communicative *speech act* S-Delegate(A,B,τ), where A is the delegator that wants to delegate a task τ to the contractor B.

This speech act is a communication command that can be viewed as a request that does not have to be accepted. In particular, an agent may not be *able* to accept as it lacks the required capabilities for performing a task. If it is accepted, it updates the belief states of the delegator and contractor [14]. As shown in Section 6, its concrete realization in the UASTech architecture is associated with a specific delegation process.

Castelfranchi and Falcone associate an overall goal with each task being delegated, dividing the task into a tuple $\tau = \langle \alpha, \phi \rangle$ consisting of a plan α specifying which actions should be executed and a goal ϕ associated with that plan. For the purpose of defining a semantics for S-Delegate, we expand this tuple by introducing an explicit set of constraints according to our previous characterization of delegation as a general concept, resulting in a task $\tau = \langle \alpha, \phi, \mathit{constraints} \rangle$. The exact structure of a plan, goal or constraint is purposely left general at this high level of abstraction but will be dealt with in detail in later sections.

We use the KARO formalism (Knowledge, Actions, Results and Opportunities [15]), an amalgam of dynamic logic and epistemic/doxastic logic augmented with several additional modal operators, to characterize the operators used in the definition of the S-Delegate speech act and to provide a formal semantics. Specifically, for the speech act to succeed, the following conditions must be satisfied (a discussion pertaining to the semantics of all non-KARO modal operators may be found in Doherty and Meyer [14]):

(1) $\mathit{Goal}_A(\phi)$ – ϕ is a goal of the delegator A.
(2) $\mathit{Bel}_A\mathit{Can}_B(\tau)$ – A believes that B can (is able to) perform the task τ. This implies that A believes that B itself believes that it can do the task: $\mathit{Bel}_A\mathit{Bel}_B(\mathit{Can}_B(\tau))$.
(3) $\mathit{Bel}_A(\mathit{Dependent}(A,B,\tau))$ – A believes it is dependent on B with respect to the task τ.
(4) $\mathit{Bel}_B\mathit{Can}_B(\tau)$ – the potential contractor B believes that it can do the task.

The following postconditions will be satisfied if the speech act is successful:

(1) $\mathit{Goal}_B(\phi)$ and $\mathit{Bel}_B\mathit{Goal}_B(\phi)$ – B has ϕ as its goal and is aware of this.
(2) $\mathit{Committed}_B(\tau)$ – B is committed to performing τ.
(3) $\mathit{Bel}_B\mathit{Goal}_A(\phi)$ – B believes that A has the goal ϕ.
(4) $\mathit{Can}_B(\tau)$ – B can do the task τ. Hence it also believes it can do the task, $\mathit{Bel}_B\mathit{Can}_B(\tau)$. Furthermore, since it has ϕ as a goal and believes it can do the task, it also intends to do the task, $\mathit{Intend}_B(\tau)$, which was a separate condition in Castelfranchi & Falcone's formalization.
(5) $\mathit{Intend}_A(\mathit{do}_B(\langle \alpha, \mathit{constraints} \rangle))$ – A intends that B should perform $\langle \alpha, \mathit{constraints} \rangle$, so we have formalized the notion of a goal to have an achievement in Castelfranchi & Falcone's informal theory to an intention to perform a task.
(6) $\mathit{MutualBel}_{AB}$("the statements above" $\wedge$ $\mathit{SociallyCommitted}(B,A,\tau)$) – there is a mutual belief between A and B that all preconditions and other postconditions mentioned hold, as well as that there is a contract between A and B, i.e. B is socially committed to A to achieve τ for A.

The *Can* predicate used in the speech act operator is a particularly important predicate. It is formally axiomatized in KARO, but in order to ground it appropriately so it reflects the actual computational processes used for delegation in the architecture, an alternative definition is required which is dependent on the task structure used in the architecture. We will therefore return to this issue in Section 4.3.

A social commitment (contract) between two agents typically induces obligations to the partners involved, depending on how the task is specified in the delegation action. This dimension of the contract affects the autonomy of the agents, in particular the contractor's autonomy. From this perspective the definition of a task $\tau = \langle \alpha, \phi, \mathit{constraints} \rangle$ is quite flexible, permitting variations that capture different levels of adjustable autonomy as well as levels of trust that can be exchanged between two agents.

For example, if α is a single elementary action with the goal ϕ implicit and correlated with the post-condition of α, the contractor has little flexibility as to how the task will be achieved. This is a form of *closed delegation* as defined by Castelfranchi & Falcone. On the other hand, if a goal ϕ is specified but α is an extensible and initially empty plan, the contractor can have a great deal of flexibility in choosing how to concretely elaborate the delegated task. This would result in a detailed plan α' extending α and satisfying the goal, a form of *open delegation* where automated planning techniques may aid in determining how the task can be performed. There are many variations between these extremes, especially considering the possibility of providing a set of explicit constraints as part of a task.

It is important to keep in mind that this characterization of delegation is not completely hierarchical. There is interaction between the delegators and contractors as to how tasks can best be achieved given the constraints of the agents involved. This is implicit in the characterization of open delegation above, though the process is not made explicit at this level of abstraction.

The pre- and post-conditions to the S-Delegate speech act are used as the formal semantic specification to the actual processes involved in the implementation of delegation in the software architecture.

2.2. *Collaborative Software Architecture Overview*

The delegation-based framework discussed above forms the basis of a concrete collaborative software architecture [7]. Currently, this architecture employs an *agent layer* (Figure 6) which encapsulates higher-level deliberative functionalities and provides a common interface for multi-agent collaboration in complex missions, including support for mission specification languages, delegation, and planning.

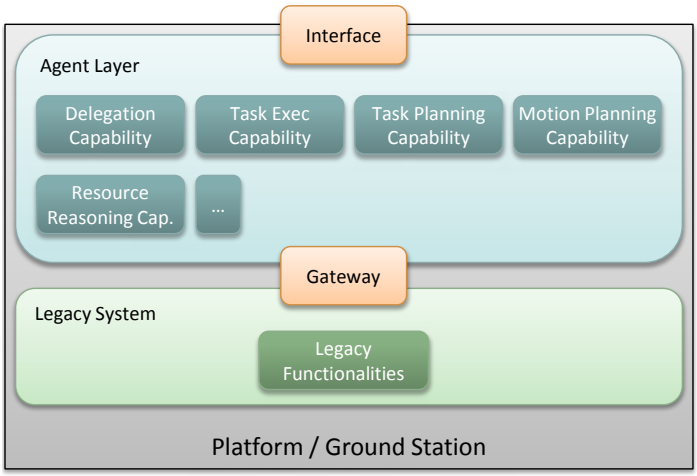

Figure 6. Agentified platform or ground control station.

As part of the agent layer, each agent has a set of conceptual *capabilities*. For example, an agent needs a *delegation capability*: It must be capable of coordinating delegation requests to and from other agents such as unmanned aircraft and ground control stations. The actual implementation of this capability must be grounded in a concrete task representation (Section 4) and will be discussed further in Section 6.

The delegation capability must also be able to determine whether the preconditions of a delegation request are satisfied. This in turn requires a *resource reasoning capability* which can determine whether the agent has the resources and ability to actually do a task as a contractor, which may require the invocation of schedulers, planners and constraint solvers. After an agent is delegated a specific task, it must eventually execute that task relative to the constraints associated with it, necessitating a *task execution capability* coordinating this execution process. Each agent also has a task planning capability in order to be able to generate or elaborate mission specifications given specific goals (Sections 5 and 7), which in turn often requires a motion planning capability [2, 16, 17] Additional capabilities, such as *chronicle recognition* to detect complex sequences of events [18, 19] and *anchoring* to ground symbolic reasoning in sensor data [20, 21], can also be present.

The agent layer also provides a common *interface* through which agents can communicate with each other, and with ground operators which can also be viewed as agents. This interface is the clearinghouse for all forms of inter-agent communication, and all requests for delegation and other types of communication pass through the interface. It also provides the external interface to a specific robotic system or ground control station.

Communication between agents is based on the exchange of messages representing *speech acts* encoded in an agent communication language, such as the FIPA ACL [22]. As mentioned earlier, delegation is one of the forms of communication concretely realized as speech acts. In addition to the formal speech act, an *interaction protocol* is required, specifying a pattern of interaction realized as a sequence of messages sent between two agents. One such protocol is defined for the messages involved in delegation, where a call-for-proposals message specifying a task and its associated constraints is sent to a potential contractor and the contractor either refuses, due to being unable to satisfy the request, or replies with a concrete proposal for how the task can be carried out (Section 6). Additional steps are then required to accept or refuse the proposal [7].

Legacy Systems. When an agent-based architecture is used together with an existing platform such as an unmanned aircraft, there may already be an existing legacy system providing a variety of lower-level functionalities such as platform-specific realizations of elementary tasks and resources. Existing interfaces to such functionalities can vary widely. The current instantiation of the architecture (Figure 6) directly supports the use of such legacy functionalities through the use of a *gateway*. The gateway must have a platform-specific implementation, but provides a common platform-independent external interface to the available legacy functionalities. In essence, this allows newly developed higher-level functionalities to be seamlessly integrated with existing systems, without the need to modify either the agent layer or the existing system. The agent layer can then be developed independently of the platforms being used.

Legacy control stations and user interfaces that human op-

erators use to interact with robotic systems are treated similarly, through the addition of an agent layer. The result is a collaborative human/robot system consisting of a number of human operators and robotic platforms each having an agent layer and possibly a legacy system, as shown in Figure 7.

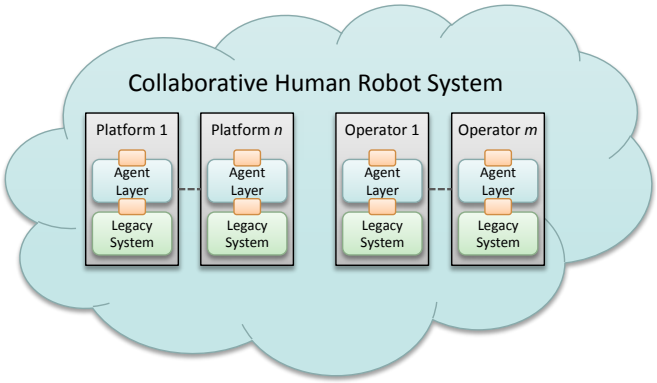

Figure 7. Overview of the collaborative human/robot system.

2.3. *Collaborative Software Architecture Implementation*

The collaborative architecture from the previous section is concretely implemented as a set of distributed processes communicating through service calls and transmitting information through a set of distributed communication channels. In early iterations, the underlying infrastructure facilitating the implementation of these general concepts was based on CORBA, while the current version is based on ROS, the Robot Operating System.

ROS (`www.ros.org`) is a convenient open-source framework for robot software development which allows interfaces and services to be clearly specified [23]. The ROS framework is multilingual with full support for C++, Lisp and Python, and each programming language has its associated client library which provides the tools and functions needed for developing ROS software.

Software written for ROS is organized into *packages* which contain nodes, libraries and configurations. *Nodes* represent computational processes in the system and are written using the client libraries. For example, many of the abstract capabilities mentioned above, including task planning and motion planning, are realized as separate ROS nodes. These nodes communicate in two ways. First, by passing structured messages on *topics* using XML-RPC where topics can be seen as named buses to which nodes can subscribe. Second, by using request/reply communication through *services*. A collaborative system consists of a number of platforms which currently share a common ROS Master, a standard ROS functionality providing registration and lookup services. For disconnected operation, the ROS Master will be federated.

Figure 8 gives an overview of some of the processes and ROS topics that are normally present in our system when applied to the LinkQuad platform. Black arrows indicate calls between processes. Each gray rectangle contains processes and topics that are explicitly associated with a particular agent. Inside this, each rounded rectangle represents a distinct functionality that is currently implemented as one or more related ROS services provided by one or more ROS nodes. Given the fundamental structure of ROS, functionality can easily be moved between nodes without affecting the remainder of the system. Therefore an exact specification at the node level is not relevant at this level of abstraction.

The *FIPA Gateway* is part of the *interface* functionality of each agent, as shown in Figure 6. This gateway listens to FIPA ACL messages on a ROS topic available to all agents in the system (the thick arrow at the top), selects those that are intended for the current platform, and forwards them to their "internal" destinations through other ROS topics.

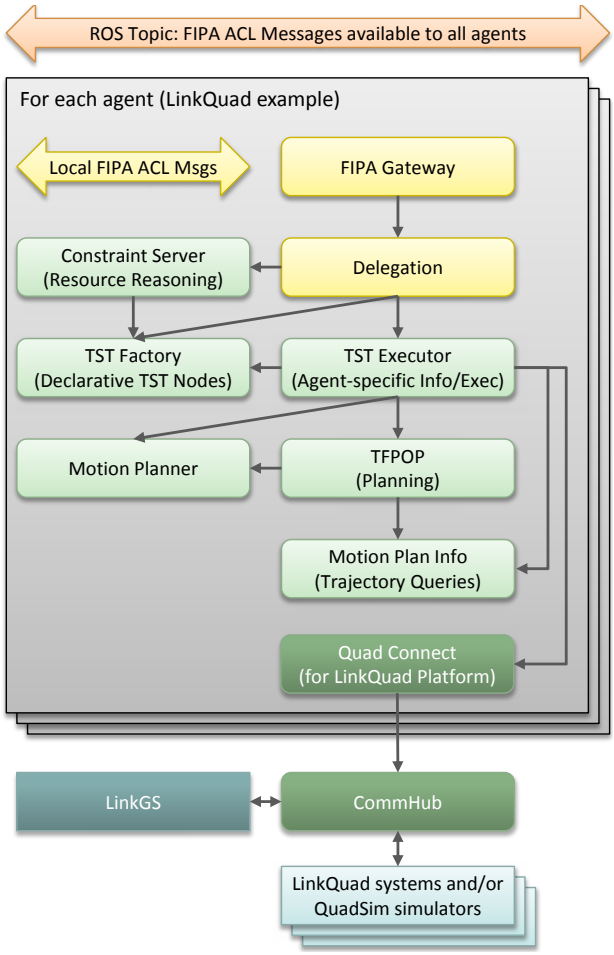

Figure 8. Overview of the ROS-based implementation.

Delegation handles delegation requests specified as FIPA speech acts. More details regarding this concrete functionality will be discussed in Section 6. Delegation also requires reasoning about whether and how the agent can satisfy the temporal and resource-related constraints associated with a task. This is implemented through the *constraint server* which is part of the resource reasoning capability mentioned earlier. The relation be-

tween constraint reasoning and delegation is also discussed in Section 6.1.

Tasks are concretely represented as Task Specification Trees (Section 4). Nodes in such trees, which are general and declarative specifications of tasks to perform, are both created by and stored in a *TST Factory*. The *TST Executor* functionality is then responsible for generating platform-specific procedural *executor* objects for any node type that a particular agent supports. Platform-specific constraints on how a certain task can be performed, such as those related to the flight envelope of the LinkQuad, are also specified here as opposed to in the more general TST Factory.

To execute a flight action, an executor may have to call a *motion planner*. This provides the ability to generate motion plans and flight trajectories using several different motion planning algorithms [24]. The related *Motion Plan Info* functionality provides the ability to estimate flight times for trajectories generated by the motion planner, which is required for the TST Executor to provide correct constraints on possible execution times for a flight task.

TFPOP (Sections 5 and 7; [25, 26]) is used as a task planner. One of its responsibilities is to expand *goal nodes* in a task specification tree into detailed task specifications, which is why it can be called from the TST Executor. TFPOP can also call the motion planner and Motion Plan Info in order to estimate whether a given flight action is feasible as part of a plan being generated, and if so, at what cost.

QuadConnect is a ROS node that converts ROS service calls to UDP-based communication with the LinkQuad system, similar to the gateway used for legacy systems. Outside the agent rectangle, the *CommHub* multiplexes communication channels to ensure that multiple programs can communicate with each LinkQuad platform simultaneously, and can forward UDP messages across serial links if desired. *LinkGS* provides a low-level interface for monitoring and debugging control loops and similar issues that do not belong at the agent level, and connects to platforms through the CommHub. Finally, CommHub can connect to the actual low-level *LinkQuad* system as well as to the corresponding *QuadSim* simulation system.

3. Mission Specification Language

Before going into details regarding concrete distributed task representation structures, we will now define a language that can be used to formally define and reason about missions at a high level of abstraction. The foundation for the language is a well-established non-monotonic logic for representing and reasoning about actions: Temporal Action Logic, TAL [27, 28]. This logic provides both clear intuitions and a formal semantics for a highly expressive class of action specifications which includes explicit support for temporal aspects, concurrent execution, and incomplete knowledge about the environment and the effects of an action, making it a suitable basis for describing the elementary actions used in realistic mission specifications. TAL has also already proven very flexible in terms of extensibility [29–31], which can be exploited to extend the logic with new constructs for *composite actions* that also capture the high-level structure of a mission [32].

Composite actions will be characterized recursively through the introduction of the new general construct "with VARS do TASK where CONS": Any composite action consists of a task TASK that should be executed in a context characterized by a set of variables VARS constrained by a set of constraints CONS. The TASK, in turn, can be an elementary TAL action or consist of a combination of composite actions using constructs such as sequence, concurrency, conditionals, (unbounded) loops, while-do, and a concurrent for-each operator. Composing actions or tasks in this manner results in a hierarchical task structure suitable for iterative decomposition during delegation, where delegation is always applied to a single composite action already existing in the task and there is a clear and comprehensible assignment of responsibilities. As will be seen below, tasks can also specify calls to named composite actions, allowing recursivity in the tasks themselves.

By allowing a set of explicit constraints to be specified locally in every elementary and composite action, we elaborate on the general definition of tasks as consisting of separate plans and goals as discussed in the previous section (the integration of goals into mission specifications will be discussed in Section 4). When specifying missions, this yields considerable advantages in terms of modularity and readability. When delegating missions in a concrete implementation, it facilitates distributing constraints to the agents that are involved in enforcing them. The ability to specify expressive constraints also permits a flexible policy of least commitment in terms of the temporal and structural constraints that are *implicit* in each task construct, as will be exemplified below.

At the mission specification level considered here, each constraint definition can be as general as a logical formula in TAL, giving it a formal semantics. For pragmatic use in a robotic architecture, a wide class of formulas can be automatically transformed into constraints processed by a constraint satisfaction solver, allowing a robotic system to formally verify the consistency of a (distributed) task through the use of (distributed) constraint satisfaction techniques (see Section 6.1).

3.1. *Temporal Action Logic*

Temporal Action Logic provides an extensible macro language, $\mathscr{L}$(ND), that supports the knowledge engineer and allows action definitions and other information to be specified at a higher abstraction level than plain logical formulas. The logic is based on the specification of scenarios represented as *narratives* in this language. Each narrative consists of a set of statements of specific types. For example, there are statement types for specifying action types with preconditions and effects, action occurrences representing the fact that a specific action occurs at a particular time, domain constraints representing general knowledge about a domain such as the emergency services domain, dependency constraints representing causal relations such as side effects specified outside of explicitly invoked actions, and partial or complete observations regarding the actual state of the world at various points in time including but not limited to the initial state.

The $\mathscr{L}$(ND) Language. We will now describe a subset of TAL which is sufficient for elementary actions in the mission examples that will be presented. The basic ontology includes parameterized features $f(\bar{x})$ that have values v at specific timepoints t, denoted by $[t]f(\bar{x}) \hat{=} v$, or over intervals, $[t,t']f(\bar{x}) \hat{=} v$. Incomplete information can be specified using disjunctions of such facts. Parameterized actions can occur at specific intervals of time, denoted by $[t_1,t_2]A(\bar{x})$. To reassign a feature to a new value, $R([t]f(\bar{x}) \hat{=} v)$ is used. Again, disjunction can be used inside $R()$ to specify incomplete knowledge about the resulting value of a feature. The value of a feature at a timepoint is denoted by $value(t,f)$.

An *action type specification* declares a named elementary action. The basic structure, which can be elaborated considerably [28], is as follows:

$$[t_1,t_2]A(\bar{v}) \rightsquigarrow (\Gamma_{pre}(t_1,\bar{v}) \rightarrow \Gamma_{post}(t_1,t_2,\bar{v})) \wedge \Gamma_{cons}(t_1,t_2,\bar{v})$$

stating that if the action $A(\bar{v})$ is executed during the interval $[t_1,t_2]$, then given that its preconditions $\Gamma_{pre}(t_1,\bar{v})$ are satisfied, its postconditions or effects, $\Gamma_{post}(t_1,t_2,\bar{v})$, will take place. Additionally, $\Gamma_{cons}(t_1,t_2,\bar{v})$ specifies logical constraints associated with the action during its execution. As an example, the following defines the elementary action **fly-to** that will later be used in an emergency response scenario: If a UAV should fly to a new position, it must initially have sufficient fuel. At the next timepoint the UAV will not be hovering, and in the interval between the start and the end of the action, the UAV will arrive and its fuel level will decrease. Finally, there is partial information about the possible flight time.

$$\begin{aligned}
&[t,t']\textbf{fly-to}(uav,x',y') \rightsquigarrow \\
&\quad [t]\ \mathsf{fuel}(uav) \geq \mathsf{fuel\text{-}usage}(uav,\mathsf{x}(uav),\mathsf{y}(uav),x',y') \rightarrow \\
&\quad R([t+1]\ \mathsf{hovering}(uav) \hat{=} \mathit{False}) \wedge \\
&\quad R([t,t']\ \mathsf{x}(uav) \hat{=} x') \wedge \\
&\quad R([t,t']\ \mathsf{y}(uav) \hat{=} y') \wedge \\
&\quad R([t,t']\ \mathsf{fuel}(uav) \hat{=} value(t,\mathsf{fuel}(uav) - \\
&\qquad\qquad \mathsf{fuel\text{-}usage}(uav,\mathsf{x}(uav),\mathsf{y}(uav),x',y'))) \wedge \\
&\quad t'-t \geq value(t,\mathsf{min\text{-}flight\text{-}time}(uav,\mathsf{x}(uav),\mathsf{y}(uav),x',y')) \wedge \\
&\quad t'-t \leq value(t,\mathsf{max\text{-}flight\text{-}time}(uav,\mathsf{x}(uav),\mathsf{y}(uav),x',y'))
\end{aligned}$$

Elementary actions will be used as the basic building blocks when we extend $\mathscr{L}$(ND) to support composite actions, and their syntax and semantics will remain the same as in TAL.

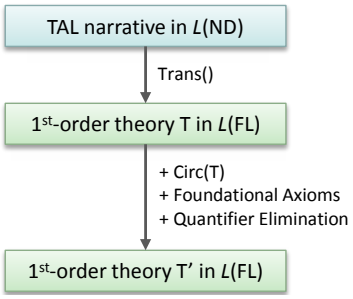

Figure 9. Relation between $\mathscr{L}$(ND) and $\mathscr{L}$(FL).

The $\mathscr{L}$(FL) language, Circumscription, and Reasoning. As shown in Figure 9, a translation function denoted by $Trans()$ translates $\mathscr{L}$(ND) expressions into $\mathscr{L}$(FL), a first-order logical language [28]. This provides a well-defined formal semantics for narratives in $\mathscr{L}$(ND). When adding new constructs to the formalism, the basic idea is to define new expression types in $\mathscr{L}$(ND) and extend the translation function accordingly. This is how composite actions have been added.

The $\mathscr{L}$(FL) language is order-sorted, supporting both types and subtypes for features and values. This is also reflected in $\mathscr{L}$(ND), where we often assume variable types are correlated to variable names – for example, uav_3 is a variable implicitly ranging over UAVs. There are a number of sorts for values $\mathscr{V}_i$, including the Boolean sort $\mathscr{B}$ with the constants $\{true,false\}$. $\mathscr{V}$ is a supersort of all such value sorts. There are a number of sorts for features $\mathscr{F}_i$, each one associated with a value sort $dom(\mathscr{F}_i) = \mathscr{V}_j$ for some j. The sort $\mathscr{F}$ is a supersort of all fluent sorts. There is also an action sort $\mathscr{A}$ and a temporal sort $\mathscr{T}$. Generally, t,t' will denote temporal variables, while $\tau,\tau',\tau_1,\ldots$ denote temporal terms. $\mathscr{L}$(FL) currently uses the following predicates, from which formulas can be defined inductively using the standard rules, connectives and quantifiers of first-order logic.

- $Holds$: $\mathscr{T} \times \mathscr{F} \times \mathscr{V}$, where $Holds(t,f,v)$ expresses that a feature f has a value v at a timepoint t, corresponding to $[t]\ f \hat{=} v$ in $\mathscr{L}$(ND).
- $Occlude$: $\mathscr{T} \times \mathscr{F}$, where $Occlude(t,f)$ expresses that a feature f is permitted to change values at time t. This is implicit in reassignment, $R([t]\ f \hat{=} v)$, in $\mathscr{L}$(ND).
- $Occurs$: $\mathscr{T} \times \mathscr{T} \times \mathscr{A}$, where $Occurs(t_s,t_e,A)$ expresses that a certain action A occurs during the interval $[t_s,t_e]$. This corresponds to $[t_s,t_e]A$ in $\mathscr{L}$(ND).

When a narrative is translated, $Trans()$ first generates the appropriate $\mathscr{L}$(FL) formulas corresponding to each $\mathscr{L}$(ND) statement. Additional foundational axioms that are used in all problem domains, such as unique names and domain closure axioms, are appended when required. Logical entailment then allows us to determine when actions must occur, but the fact that they *cannot* occur at other times than explicitly stated is not explicitly stated and therefore not logically entailed by the translation. This problem is handled in a general manner through filtered circumscription, which also ensures that fluents can change values only when explicitly affected by an action or dependency constraint [28].

Although the filtered circumscription axioms used by TAL are second-order formulas, the structure of $\mathscr{L}$(ND) statements ensures that they are reducible to equivalent first-order formulas, a reduction that can often be performed through predicate completion. Therefore, classical first-order theorem proving techniques can be used for reasoning about TAL narratives [28]. In the context of unmanned systems, however, the logic will primarily be used to ensure a correct semantics for the specification language that is closely correlated to the implementation. Using this language for delegation and execution will not require theorem proving on board.

3.2. *Composite Actions for Mission Specifications*

We now extend $\mathscr{L}$(ND) to support *composite action type specifications*, which declare named composite actions. This is useful in order to define a library of meaningful composite actions to be used in mission specifications. Each specification is of the form

$$[t,t']\underline{\mathsf{comp}}(\bar{v}) \rightsquigarrow A(t,t',\bar{v})$$

where $\underline{\mathsf{comp}}(\bar{v})$ is a *composite action term* such as $\underline{\mathsf{monitor\text{-}pattern}}(x,y,dist)$, consisting of an action name and a list of parameters, and $A(t,t',\bar{v})$ is a *composite action expression* where only variables in $\{t,t'\} \cup \bar{v}$ may occur free. A composite action expression (C-ACT), in turn, allows actions to be composed at a high level of abstraction using familiar programming language constructs such as sequences $(A;B)$, concurrency $(A \mid\mid B)$, conditions and loops. The syntax for composite actions is defined as follows:

C-ACT ::= $[\tau,\tau']$ with $\bar{x}$ do TASK where ϕ
TASK ::= $[\tau,\tau']$ ELEM-ACTION-TERM |
$[\tau,\tau']$ COMP-ACTION-TERM |
(C-ACT; C-ACT) |
(C-ACT || C-ACT) |
if $[\tau]\,\psi$ then C-ACT else C-ACT |
while $[\tau]\,\psi$ do C-ACT |
foreach $\bar{x}$ where $[\tau]\,\psi$ do conc C-ACT

where $\bar{x}$ is a potentially empty sequence of variables (where the empty sequence can be written as ε), ϕ is a TAL formula representing a set of constraints, ELEM-ACTION-TERM is an elementary action term such as **fly-to**(uav,x,y), COMP-ACTION-TERM is a composite action term, and $[\tau]\,\psi$ is a TAL formula referring to facts at a single timepoint τ. For brevity, omitting "with $\bar{x}$ do" is considered equivalent to specifying the empty sequence of variables and omitting "where ϕ" is equivalent to specifying "where TRUE". Note that the ; and || constructs are easily extended to allow an arbitrary number of actions, as in $(A;B;C;D)$.

Like elementary actions, every composite action C-ACT is annotated with a temporal execution interval. This also applies to each "composite sub-action". For example,

$[t_1,t_2]$with uav,t_3,t_4,t_5,t_6 do
$\big([t_3,t_4]$**fly-to**$(uav,x,y);[t_5,t_6]$**collect-video**$(uav,x,y)\big)$
where $[t_1]$has-camera(uav)

denotes a composite action where one elementary action takes place within the interval $[t_3,t_4]$, the other one within the interval $[t_5,t_6]$, and the entire sequence within $[t_1,t_2]$.

The with-do-where construct provides a flexible means of constraining variables as desired for the task at hand. In essence, "$[t_1,t_2]$with $\bar{x}$ do TASK where ϕ" states that there exists an instantiation of the variables in $\bar{x}$ such that the specified TASK (which may make use of $\bar{x}$ as illustrated above) is executed within the interval $[t_1,t_2]$ in a manner satisfying ϕ. The constraint ϕ may be a combination of temporal, spatial and other types of constraints. Above, this constraint is used to ensure the use of a *uav* that has a camera rather than an arbitrary *uav*.

As we aim to maximize temporal flexibility, the sequence operator (;) does not implicitly constrain the two actions **fly-to** and **collect-video** to cover the entire temporal interval $[t_1,t_2]$. Instead, the actions it sequentializes are only constrained to occur *somewhere within* the execution interval of the composite action, and gaps are permitted between the actions – but all actions in a sequence must occur in the specified order without overlapping in time. This is formally specified in the *TransComp* function below. Should stronger temporal constraints be required, they can be introduced in a where clause. For example, $t_1 = t_3 \wedge t_4 = t_5 \wedge t_6 = t_2$ would disallow gaps in the sequence above. Also, variations such as gapless sequences can easily be added to the language if desired.

Extending $\mathscr{L}$(FL) with Fixpoints. In order to translate composite actions into $\mathscr{L}$(FL), we need the ability to represent loops, recursion and inductive definitions. Since this is strictly outside the expressivity of a first-order language, $\mathscr{L}$(FL) is extended into the language $\mathscr{L}(\mathrm{FL}_{\mathrm{FP}})$ which allows *fixpoint expressions* of the form $\textsc{Lfp}\, X(\bar{x}).\big[\Gamma(X,\bar{x},\bar{z})\big]$, where all occurrences of X in Γ must be positive. The corresponding version of TAL is denoted by TALF. Fixpoint logic [33] strikes a nice balance between first-order and second-order logic, with an increase in expressivity conservative enough to still allow relatively efficient inference techniques. In particular, Doherty, Kvarnström and Szalas [32] present a proof system which is complete relative to a specific class of arithmetical structures that is useful for robotic applications. Furthermore, TAL structures that are bounded in the sense that there is a finite number of fluents and a finite maximum timepoint can be represented as deductive databases and queried in polynomial time.

Intuitively, the fixpoint expression $\textsc{Lfp}\, X(\bar{x}).\big[\Gamma(X,\bar{x},\bar{z})\big]$ represents the predicate that has the *smallest* extension that satisfies $X(\bar{x}) \leftrightarrow \Gamma(X,\bar{x},\bar{z})$. For example, the expression $\textsc{Lfp}\ \mathsf{path}(n_1,n_2).\big[\mathsf{edge}(n_1,n_2) \vee \exists n.(\mathsf{path}(n_1,n) \wedge \mathsf{edge}(n,n_2))\big]$ represents the transitive closure of an edge predicate, which cannot be defined using only first-order logic. The formula $\big(\textsc{Lfp}\ \mathsf{path}(n_1,n_2).\big[\mathsf{edge}(n_1,n_2) \vee \exists n.(\mathsf{path}(n_1,n) \wedge \mathsf{edge}(n,n_2))\big]\big)(A,B)$ applies this predicate to determine whether there is a path between the specific nodes A and B in a graph.

Formally, the meaning of $\textsc{Lfp}\, X(\bar{x}).\big[\Gamma(X,\bar{x},\bar{z})\big]$ is provided by the following Kleene characterization of fixpoints:

$$\textsc{Lfp}\, X(\bar{x}).\big[\Gamma(X,\bar{x},\bar{z})\big] \equiv \bigvee_{i\in\omega} \Gamma^i(\mathit{false},\bar{x},\bar{z}), \quad \text{where}$$

$$\Gamma^i(\mathit{false},\bar{x},\bar{z}) \stackrel{\text{def}}{=} \begin{cases} \mathit{false} & \text{for } i = 0 \\ \Gamma(\Gamma^{i-1}(\mathit{false},\bar{x},\bar{z}),\bar{x},\bar{z}) & \text{for } i > 0. \end{cases}$$

In the finite case this corresponds to an iterative procedure: First let $X(\bar{x})$ be false for all $\bar{x}$, corresponding to $i = 0$ above. Then repeatedly generate a new definition where $X(\bar{x})$ true iff $\Gamma(X,\bar{x},\bar{z})$ is true when evaluated with the "old" definition of X generated in the *previous* step, corresponding to $i > 0$ above. Since X only occurs positively in $\Gamma(X,\bar{x},\bar{z})$, its extension will grow monotonically. Continue updating X in this manner until its definition no longer changes, at which point a fixpoint has been reached.

Translating Composite Actions into $\mathscr{L}(\mathrm{FL}_{\mathrm{FP}})$. A formal semantics for composite actions can now be defined by extending the standard TAL translation function *Trans*() from Doherty and Kvarnström [28] and using $\mathscr{L}(\mathrm{FL}_{\mathrm{FP}})$, with a standard logical fixpoint semantics, as the target language. The extended *Trans*()

function calls $TransComp(\tau,\tau',T)$ to translate each task T according to the intended meaning that T occurs *somewhere* within the interval $[\tau,\tau']$.

$$Trans\big([\tau,\tau']\textsf{with } \bar{x} \textsf{ do } T \textsf{ where } \phi\big) \stackrel{\text{def}}{=} \exists \bar{x}\big[TransComp(\tau,\tau',T) \wedge Trans\big(\phi\big)\big]$$

If the task to be translated is a call to a named elementary action $\textbf{elem}(\bar{v})$, then *TransComp*() calls the standard $Trans\big(\big)$ function. Calls to named composite actions are discussed later.

$$TransComp(\tau,\tau',[\tau_1,\tau_2]\textbf{elem}(\bar{v})) \stackrel{\text{def}}{=} Trans\big([\tau_1,\tau_2]\textbf{elem}(\bar{v})\big) \wedge \tau \le \tau_1 < \tau_2 \le \tau'$$

Two potentially concurrent actions are simply constrained to occur within the given interval $[\tau,\tau']$. A sequence of two actions must additionally occur in the stated order. An if/then/else task is translated into a conjunction of conditionals, where both the timepoint τ_c at which the condition is checked and the execution interval of the selected action (A_1 or A_2) must be within $[\tau,\tau']$.

$$\begin{aligned}&TransComp(\tau,\tau',\big([\tau_1,\tau_2]A_1 \;||\; [\tau_3,\tau_4]A_2\big)) \stackrel{\text{def}}{=}\\ &\quad Trans\big([\tau_1,\tau_2]A_1\big) \wedge Trans\big([\tau_3,\tau_4]A_2\big) \wedge\\ &\qquad \tau \le \tau_1 \le \tau_2 \le \tau' \wedge \tau \le \tau_3 \le \tau_4 \le \tau'\end{aligned}$$

$$\begin{aligned}&TransComp(\tau,\tau',\big([\tau_1,\tau_2]A_1;[\tau_3,\tau_4]A_2\big)) \stackrel{\text{def}}{=}\\ &\quad Trans\big([\tau_1,\tau_2]A_1\big) \wedge Trans\big([\tau_3,\tau_4]A_2\big) \wedge\\ &\qquad \tau \le \tau_1 \le \tau_2 \le \tau_3 \le \tau_4 \le \tau'\end{aligned}$$

$$\begin{aligned}&TransComp(\tau,\tau',\textsf{if } [\tau_c]F \textsf{ then } [\tau_1,\tau_2]A_1 \textsf{ else } [\tau_3,\tau_4]A_2) \stackrel{\text{def}}{=}\\ &\quad (Trans\big([\tau_c]F\big) \rightarrow Trans\big([\tau_1,\tau_2]A_1\big)) \wedge\\ &\quad (Trans\big([\tau_c]\neg F\big) \rightarrow Trans\big([\tau_3,\tau_4]A_2\big)) \wedge\\ &\quad \tau \le \tau_c \le \tau' \wedge \tau_c \le \tau_1 \le \tau_2 \le \tau' \wedge \tau_c \le \tau_3 \le \tau_4 \le \tau'\end{aligned}$$

A *concurrent foreach* statement allows a variable number of actions to be executed concurrently. An example is given in the next section, where all available UAVs with the ability to scan for injured people should do so in parallel. Below, $\bar{x}$ is a non-empty sequence of value variables. For all instantiations of $\bar{x}$ satisfying $[\tau_c]F(\bar{x})$, there should be an interval within $[\tau_1,\tau_2]$ where the composite action $A(\bar{x})$ is executed.

$$\begin{aligned}&TransComp(\tau,\tau',\textsf{foreach } \bar{x} \textsf{ where } [\tau_c]F(\bar{x}) \textsf{ do conc } [\tau_1,\tau_2]A(\bar{x})) \stackrel{\text{def}}{=}\\ &\quad \forall \bar{x}[Trans\big([\tau_c]F(\bar{x})\big) \rightarrow Trans\big([\tau_1,\tau_2]A(\bar{x})\big)] \wedge\\ &\qquad \tau \le \tau_c \le \tau_1 \le \tau_2 \le \tau'\end{aligned}$$

A while loop is translated into a least fixpoint. Informally, the LFP parameter u represents the time at which the previous iteration ended, and is initially given the value τ as seen in the final line below. In each iteration the temporal variable t_c is bound to the timepoint at which the iteration condition F is tested, which must be at least u and at most τ'. If the condition holds, the variables $[t_1,t_2]$ are bound to an interval where the inner action A is executed (similarly constrained to be in $[t_c,\tau']$), the action occurs, and the next iteration may start no earlier than t_2, specified by $X(t_2)$. If the condition does not hold, no new iteration is implied by the formula and a fixpoint has been reached.

$$\begin{aligned}&TransComp(\tau,\tau',\textsf{while } [t_c]F \textsf{ do } [t_1,t_2]A) \stackrel{\text{def}}{=}\\ &\quad \tau \le \tau' \wedge \textsc{Lfp}\ X(u).\big[\exists t_c[\\ &\qquad u \le t_c \le \tau' \wedge (Trans\big([t_c]F\big) \rightarrow\\ &\qquad \exists t_1,t_2[t_c \le t_1 \le t_2 \le \tau' \wedge Trans\big([t_1,t_2]A\big) \wedge X(t_2)]\\ &\quad]\big](\tau)\end{aligned}$$

Assume a composite action is named using a statement such as $[t,t']\underline{\textsf{comp}}(\bar{x}) \rightsquigarrow A(t,t',\bar{x})$. A named action can be called in two places: As part of a composite action expression, where one composite action calls another, and at the "top level" of a narrative, where one states that a specific composite action occurs. We therefore extend both *Trans* and *TransComp*:

$$\begin{aligned}&Trans\big([\tau_1,\tau_2]\underline{\textsf{comp}}(\bar{a})\big) \stackrel{\text{def}}{=}\\ &\quad \textsc{Lfp}\ Y(t,t',\bar{x}).\big[Trans\big(A'(t,t',\bar{x})\big)\big](\tau_1,\tau_2,\bar{a})\\ &TransComp(\tau,\tau',[\tau_1,\tau_2]\underline{\textsf{comp}}(\bar{a})) \stackrel{\text{def}}{=}\\ &\quad Trans\big([\tau_1,\tau_2]\underline{\textsf{comp}}(\bar{a})\big) \wedge \tau \le \tau_1 \le \tau_2 \le \tau'\end{aligned}$$

where $A'(t,t',\bar{x})$ is $A(t,t',\bar{x})$ with all occurrences of $\underline{\textsf{comp}}$ replaced with Y. This use of fixpoint expressions in the translation of the composite action permits direct recursion: An action may contain a nested conditional call to itself, which is transformed into a "call" to Y, leading to another "iteration" in the fixpoint. To fully support mutually recursive action definitions, *simultaneous fixpoints* are required. We omit this here for brevity and readability.

As a result of this translation, a composite action is a theory in $\mathscr{L}(\text{FL}_{\text{FP}})$. Questions about missions thereby become queries relative to an inference mechanism, allowing operators to analyze mission properties both during pre- and post-mission phases.

Possible Extensions. We have now defined a small core of important composite action constructs. This is not an exhaustive list, and additional constructs can easily be added. For example, to wait for a formula ϕ to become true, we can define $[\tau,\tau']\textsf{wait-for}(\phi)$ as $Trans\big(\forall u[\tau \le u < \tau' \rightarrow [u]\neg\phi] \wedge [\tau']\phi \wedge \tau' \ge \tau\big)$: The formula is false at all timepoints in $[\tau,\tau')$ and is true exactly at τ', which cannot be before τ. This ensures that wait-for terminates at the first timepoint $\tau' \ge \tau$ where ϕ is true.

3.3. *Composite Actions in the Fukushima Scenario*

A mission specification can now be expressed as a composite action in the extended version of TAL described above. We will exemplify this concretely using a scenario related to the tsunami in Japan in 2011. Specifically, we will consider several composite actions that can be useful for the problem of information gathering near the Fukushima Daiichi nuclear plant, where an emergency response unit could use a team of unmanned aircraft to assess the situation, monitor radiation levels, and locate injured people in a regular grid around a damaged reactor. Our focus is on demonstrating the $\mathscr{L}$(ND) composite action constructs and some aspects of the composite actions below are simplified for expository reasons.

We assume the existence of a set of elementary actions whose meaning will be apparent from their names and from the

explanations below: **hover-at**, **fly-to**, **monitor-radiation**, **collect-video**, and **scan-cell**. Each elementary action is assumed to be defined in standard TAL and to provide suitable preconditions, effects, resource requirements and (completely or incompletely specified) durations. For example, only a UAV with suitable sensors can execute **monitor-radiation**.

In the following composite action, a UAV hovers at a location $(x_{\text{uav}}, y_{\text{uav}})$ while using its on-board sensors to monitor radiation and collect video at $(x_{\text{targ}}, y_{\text{targ}})$.

$$
\begin{array}{l}
[t,t'] \underline{\text{monitor-single}}(uav, x_{\text{uav}}, y_{\text{uav}}, x_{\text{targ}}, y_{\text{targ}}) \rightsquigarrow \\
[t,t'] \text{with } t_1,t_2,t_3,t_4,t_5,t_6 \text{ do } (\\
\quad [t_1,t_2] \textbf{hover-at}(uav, x_{\text{uav}}, y_{\text{uav}}) \;||\; \\
\quad [t_3,t_4] \textbf{monitor-radiation}(uav, x_{\text{targ}}, y_{\text{targ}}) \;||\; \\
\quad [t_5,t_6] \textbf{collect-video}(uav, x_{\text{targ}}, y_{\text{targ}}) \\
) \text{ where } [t]\text{surveil-equipped}(uav) \wedge \text{radiation-hardened}(uav) \wedge \\
\qquad t_1 = t_3 = t_5 = t \wedge t_2 = t_4 = t_6 = t'
\end{array}
$$

The first part of the constraint specified in the *where* clause ensures that a UAV involved in a monitoring action is equipped for surveillance. Furthermore, as we are being careful at this point, we only want to use radiation-hardened aircraft (in addition to the conditions placed on **monitor-radiation**, which include the existence of radiation sensors). The temporal constraints model a requirement for these particular actions to be synchronized in time and for the UAV to hover in a stable location throughout the execution of monitor-single. These constraints could easily be relaxed, for example by stating that hovering occurs throughout the action but monitoring occurs in a sub-interval.

The following action places four UAVs in a diamond pattern to monitor a given location such as a nuclear reactor at a given distance, counted in grid cells. The UAVs involved are not specified as parameters to the monitoring action, but are chosen freely among available UAVs. Note, however, that the constraints placed on parameters by sub-actions will apply at this higher level as well. Thus, as for monitor-single, all UAVs involved in this action must be equipped for surveillance and carry radiation sensors.

$$
\begin{array}{l}
[t,t'] \underline{\text{monitor-pattern}}(x, y, dist) \rightsquigarrow \\
[t,t'] \text{with } s_1,\ldots,w_4, uav_1, uav_2, uav_3, uav_4 \text{ do } (\\
\quad ([s_1,s_2]\ \textbf{fly-to}(uav_1, x+dist, y); \\
\quad\ [s_3,s_4]\ \underline{\text{monitor-single}}(uav_1, x+dist, y, x, y)) \;||\; \\
\quad ([u_1,u_2]\ \textbf{fly-to}(uav_2, x-dist, y); \\
\quad\ [u_3,u_4]\ \underline{\text{monitor-single}}(uav_2, x-dist, y, x, y)) \;||\; \\
\quad ([v_1,v_2]\ \textbf{fly-to}(uav_3, x, y+dist); \\
\quad\ [v_3,v_4]\ \underline{\text{monitor-single}}(uav_3, x, y+dist, x, y)) \;||\; \\
\quad ([w_1,w_2]\ \textbf{fly-to}(uav_4, x, y-dist); \\
\quad\ [w_3,w_4]\ \underline{\text{monitor-single}}(uav_4, x, y-dist, x, y))) \\
\text{where} \\
\quad s_3 = u_3 = v_3 = w_3 \wedge s_4 = u_4 = v_4 = w_4 \wedge \\
\quad s_4 - s_3 \geq \text{minduration}
\end{array}
$$

Four sequences are executed in parallel. Within each sequence, a specific UAV flies to a suitable location and then monitors the target. We require the target to be monitored simultaneously by all four UAVs ($s_3 = u_3 = v_3 = w_3$ and $s_4 = u_4 = v_4 = w_4$), while $s_4 - s_3 \geq \text{minduration}$ ensures this is done for at least the specified duration. As flying does not need to be synchronized, the intervals for the **fly-to** actions are only constrained *implicitly* through the definition of a sequence. For example, the translation ensures that $t \leq s_1 \leq s_2 \leq s_3 \leq s_4 \leq t'$, so that each **fly-to** must end before the corresponding monitor-single.

All grid cells must also be scanned for injured people. The following generic action makes use of all available UAVs with the proper capabilities, under the assumption that each such UAV has been assigned a set of grid cells to scan. An assignment could be generated by another action or provided as part of the narrative specification. For clarity, we include several clauses (with ε do, where TRUE) that could easily be omitted.

$$
\begin{array}{l}
[t,t'] \ \underline{\text{scan-with-all-uavs}}() \rightsquigarrow \\
[t,t'] \text{ with } \varepsilon \text{ do} \\
\qquad \text{foreach } uav \text{ where } [t]\text{can-scan}(uav) \text{ do conc} \\
\qquad\quad [t,t'] \text{ with } u,u' \text{ do} \\
\qquad\qquad [u,u'] \ \underline{\text{scan-for-people}}(uav) \\
\qquad\qquad\ \text{where TRUE} \\
\qquad \text{where TRUE}
\end{array}
$$

As shown below, each UAV involved in this task iterates while there remains at least one grid cell (x,y) that it has been assigned ("owns") and that is not yet scanned. In each iteration the variables (x',y') declared in the nested with clause range over arbitrary coordinates, but the associated where clause ensures that only coordinates that belong to the given UAV and that have not already been scanned can be selected. Also in each iteration, the variable t_c is bound to the time at which the constraint condition is tested and u,u' are bound to the timepoints at which the inner composite action is performed. The repeated use of u,u' is intentional: The elementary **scan-cell** action will occur over exactly the same interval as the enclosing composite action construct. Note also that u,u' are automatically declared by the while statement and do not have to occur in a with clause. This is necessary as they will have different values in each iteration and must be declared inside the fixpoint translation of while tasks as specified earlier.

$$
\begin{array}{l}
[t,t'] \ \underline{\text{scan-for-people}}(uav) \rightsquigarrow \\
[t,t'] \text{ with } \varepsilon \text{ do} \\
\qquad \text{while } [t_c]\ \exists x,y[\text{owns}(uav,x,y) \wedge \neg\text{scanned}(x,y)] \text{ do} \\
\qquad\quad [u,u'] \text{ with } x',y' \text{ do} \\
\qquad\qquad\ [u,u']\textbf{scan-cell}(uav,x',y') \\
\qquad\qquad\ \text{where } [t_c]\text{owns}(uav,x',y') \wedge \neg\text{scanned}(x',y') \\
\qquad \text{where TRUE}
\end{array}
$$

Finally, we can define a small mission to occur within the interval $[0,1000]$, where scanning may utilize the entire interval while the grid cell $(20,25)$ is monitored at a distance of 3 cells and must terminate before time 300.

$$
\begin{array}{l}
[0,1000]([0,1000]\underline{\text{scan-with-all-uavs}}() \;||\; \\
\qquad\qquad [0,300]\underline{\text{monitor-pattern}}(20,25,3))
\end{array}
$$

It should be emphasized that in the expected case, the task of generating specifications of this kind would be aided by libraries of predefined domain-related actions as well as by user interfaces adapted to the task at hand. The structure and high-level nature of the language remains important when ensuring that

these tools and their output are both correct and comprehensible to a human operator inpecting a mission definition.

4. Task Specification Trees

The composite action formalism defined in the previous section allows us to view a mission as a complex multi-agent task that can be recursively constructed from simpler or smaller tasks. This results in an abstract, formally grounded and very expressive mission specification language. However, it is equally essential to tightly ground mission specifications into the actual operations of a robotic system. We would therefore like to be able to map and compile them into a form of task structure that is pragmatically usable on board agent-based autonomous unmanned systems. This structure should be used both in delegation, as a concrete task representation for the S-Delegate speech act, and for execution.

We have proposed *Task Specification Trees* (TSTs) for this purpose and applied them in a number of deployable UAV systems [34, 35]. Each node in such a tree is a specification of an action, a control structure, or a goal.

Example 4.1. Consider a small scenario where the mission is that two UAVs should concurrently scan the areas Area_A and Area_B, after which the first UAV should fly to Dest_4 (Figure 10). The corresponding Task Specification Tree (Figure 11) uses three *elementary action nodes* (marked E), corresponding to two elementary actions of type **scan-area** and one of type **fly-to**. Furthermore, it requires a *concurrent node* (marked C) specifying that the **scan-area** actions can be performed concurrently, as well as a *sequential node* (marked S). Further explanations will be given below.

Figure 10. Map of example mission area.

A task specification tree is in itself purely declarative, defining what should be achieved and providing parameters and constraints for tasks. For example, a sequence node declaratively specifies that its children should be sequentially executed, while a **fly-to** node would specify that an aircraft should fly to a specific location, possibly with parameters such as speed and altitude. It is up to each platform to associate each node type with an *executor*, an executable procedural representation specifying *how* the corresponding task should be performed. This provides a clear separation between task specifications and platform-specific execution details. Note that executors are required even for *control nodes* or *structural nodes* , though such executors may be identical across platforms. For example, a sequential node requires an executor that procedurally ensures its children are executed in sequential order.

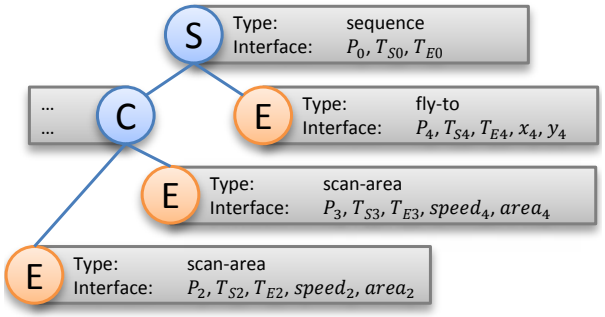

Figure 11. Example Task Specification Tree.

Since many agents can cooperate in performing a mission, the task structure used must be both *sharable* and *distributable*. This is achieved by allowing nodes to reside on multiple platforms and edges to cross between platforms. Each node in the tree therefore represents a task that can be delegated to another agent, which may be an unmanned platform or (in a mixed-initiative setting) a human working through an agentified interface. More concretely, the current implementation uses node identities consisting of a platform ID combined with a platform-local node ID, together with ROS services allowing TST-related communication across platforms. For example, when delegating a tree to another platform, what is transmitted is not the tree but the node identity. The receiving delegation functionality can then query the delegator for information about the TST, and if desired, it can generate its own local copy linked into the original tree.

Parameters and Constraints. As illustrated in Figure 11, the task corresponding to a node is identified through the node type (such as sequence or **fly-to**) together with a set of *node parameters* corresponding directly to "with $\vec{x}$" in a composite action. Together, these parameters are called the *node interface*. The node interface always contains a platform (agent) assignment parameter, usually denoted by P_i. This identifies the agent that will execute the action and corresponds to the *uav* agent parameter used previously for actions in the high-level mission specification language. The interface also always contains two parameters for the start and end times of the task, usually denoted by T_{Si} and T_{Ei}, respectively.

Additional parameters may also be present, such as a speed parameter and an area parameter for a **scan-area** action. Actual arguments, such as the fact that the areas to scan are Area_A and

$\mathsf{Area_B}$, are not shown in the figure.

Also not shown explicitly in the figure is the fact that each node can be associated with a set of constraints, called *node constraints*. These constraints correspond directly to the **where** clause in a TAL composite action and can be used to constrain the parameters of the same node as well as all parameters of ancestor nodes in the tree. Constraints can also express precedence relations and organizational relations between the nodes in the TST that are not captured by the use of specific control nodes. Together the constraints form a constraint network where the node parameters function as constraint variables.

Note that constraining node parameters implicitly constrains the degree of autonomy of an agent, as it reduces the space of possibilities that the agent can choose from. Also, both human and robotic agents may take the initiative and set the values of the parameters in a TST before or during execution. This provides support for one form of mixed-initiative interaction.

TSTs and Goals. Castelfranchi and Falcone [9, 10] permit a single goal to be associated with a task. To be able to specify desired goals to be achieved at multiple points in a mission, we extend this by specifying *goal nodes* that can be embedded at arbitrary points in a task specification tree. This allows more fine-grained control over some aspects of a mission while permitting an agent to call an on-board task planner to determine how to achieve other aspects of the same mission. Depending on the constraints that are used, a contractor can be given a great deal of freedom, thereby increasing its permitted level of autonomy. It can then satisfy the goals using any means at its disposal.

When a goal node is in the process of being delegated to a particular agent, the agent cannot accept the delegation until it has determined that it can find a plan achieving the goal (Section 5) and that it can either execute the plan itself or find subcontractors that can. Once a plan has been generated it must be integrated into the task specification tree, implying that the task representation must be *dynamically expandable*. Goal nodes are therefore allowed to add children to themselves, after which they act as sequence nodes. This is one example of open delegation where a contractor elaborates a task α into a detailed plan α' as discussed earlier. Once the plan has been integrated, the full TST corresponds to a complete mission specification.

4.1. *Syntax: Describing Task Specification Trees*

Though TSTs consist of nodes and edges, describing them in a text-based language is often more convenient. The syntax in Figure 12 is used for this purpose and was in fact defined for pragmatic reasons before the TAL-based mission specification language was developed.

As seen in the figure, there is a close but not perfect correspondence between the TST description language and the mission specification language. For example, each TST construct corresponds to a specific parameterized node. All such nodes must be explicitly named in order to allow name-based references, which was not the case for a TAL composite action construct such as a sequence. The parameters, VARS, specify the node interface which can be accessed from the outside and correspond directly to a combination of the temporal interval and the with clause of a composite action. As in composite actions, these can be constrained relative to each other using constraints in a **where** clause.

Some node types use conditions (COND), which can be represented as FIPA ACL queries. The concrete content of such a query can be realized as an arbitrary TAL formula sent to a theorem proving module such as the deductive database-based inference mechanism presented in Doherty, Kvarnström and Szalas [32]. It is also possible to retain the same semantics while syntactically restricting formulas to use subsets of TAL that can be handled through constraint solving or other efficient querying techniques.

```
TST    ::= NAME '(' VARS ')' '='
                (with VARS)? TASK (where CONS)?
TSTS   ::= TST | TST ';' TSTS
TASK   ::= ACTION | GOAL | call NAME '(' ARGS ')' |
           sequence TSTS | concurrent TSTS |
           if [TIME] COND then TST else TST |
           while [TIME] COND TST |
           foreach VARS where [TIME] COND do conc TST
VAR    ::= <variable name>
VARS   ::= VAR | VAR ',' VARS
ARG    ::= VAR | VALUE
ARGS   ::= ARG | ARG ',' ARGS
CONS   ::= <constraint> | <constraint> and CONS
VALUE  ::= <value>
TIME   ::= <temporal expression>
NAME   ::= <node name>
COND   ::= <FIPA ACL query message
             requesting the value of a boolean expression>
GOAL   ::= <goal statement name(x̄)>
ACTION ::= <elementary action call name(x̄)>
```

Figure 12. Task Specification Tree syntax

Example. Consider a small scenario where the mission is to first scan $\mathsf{Area_A}$ and $\mathsf{Area_B}$, and then fly to $\mathsf{Dest_4}$ (Figure 11). In the associated TST, nodes marked S and C are sequential and concurrent nodes, respectively, while nodes marked E are elementary action nodes. The corresponding TST specification associates each node with a task name τ_i. There are two composite actions, one sequential (τ_0) and one concurrent (τ_1), and three elementary actions of type **scan-area** and **fly-to**:

$\tau_0(T_{S_0},T_{E_0}) =$
with $T_{S_1},T_{E_1},T_{S_4},T_{E_4}$ **sequence**
  $\tau_1(T_{S_1},T_{E_1}) =$
  with $T_{S_2},T_{E_2},T_{S_3},T_{E_3}$ **concurrent**
    $\tau_2(T_{S_2},T_{E_2})$ = **scan-area**$(T_{S_2},T_{E_2},Speed_2,Area_A)$,
    $\tau_3(T_{S_3},T_{E_3})$ = **scan-area**$(T_{S_3},T_{E_3},Speed_3,Area_B)$
  where $cons_{\tau_1}$,
  $\tau_4(T_{S_4},T_{E_4})$ = **fly-to**$(T_{S_4},T_{E_4},Dest_4x, Dest_4y)$
where $cons_{\tau_0}$

$cons_{\tau_0} = T_{S_0} \leq T_{S_1} \leq T_{E_1} \leq T_{S_4} \leq T_{E_4} \leq T_{E_0}$
$cons_{\tau_1} = T_{S_1} \leq T_{S_2} \leq T_{E_2} \leq T_{E_1}$ **and** $T_{S_1} \leq T_{S_3} \leq T_{E_3} \leq T_{E_1}$

4.2. *Semantics: Connecting TSTs to* TALF

Since task specification trees serve as an executable counterpart of mission specifications, there must be a close connection between the semantics of a TALF composite action and the corresponding TST. The semantics of a TST is in fact defined in terms of this connection using a bidirectional translation function as illustrated in Figure 4.

We will begin by defining the translation in the direction from TALF composite actions into TSTs. Temporal constraints implicit in a composite action must then be made explicit and must be "lifted" into the nearest *enclosing* **with** clause in a TST, to make them explicitly available to a constraint solver. The *TransTST*() function therefore relies on a *TransTask*() function, corresponding closely to *TransComp*(), which returns a tuple containing both a TST TASK and a constraint.

$$\begin{aligned}
&\mathit{TransTask}(\tau,\tau',[\tau_1,\tau_2]\mathbf{elem}(\bar{v})) \stackrel{\text{def}}{=}\\
&\quad\langle \mathbf{elem}(\tau_1,\tau_2,\bar{v}), \tau \le \tau_1 \le \tau_2 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',[\tau_1,\tau_2]\underline{\mathsf{comp}}(\bar{v})) \stackrel{\text{def}}{=}\\
&\quad\langle \mathbf{call}\ \underline{\mathsf{comp}}(\tau_1,\tau_2,\bar{v}), \tau \le \tau_1 \le \tau_2 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',([\tau_1,\tau_2]A_1;[\tau_3,\tau_4]A_2)) \stackrel{\text{def}}{=}\\
&\quad\langle(\mathbf{sequence}\ \mathit{TransTST}([\tau_1,\tau_2]A_1), \mathit{TransTST}([\tau_3,\tau_4]A_2)),\\
&\qquad \tau \le \tau_1 \le \tau_2 \le \tau_3 \le \tau_4 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',([\tau_1,\tau_2]A_1 \mid\mid [\tau_3,\tau_4]A_2)) \stackrel{\text{def}}{=}\\
&\quad\langle(\mathbf{concurrent}\ \mathit{TransTST}([\tau_1,\tau_2]A_1), \mathit{TransTST}([\tau_3,\tau_4]A_2)),\\
&\qquad \tau \le \tau_1 \le \tau_2 \le \tau' \wedge \tau \le \tau_3 \le \tau_4 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',\mathsf{if}\ [\tau_c]F\ \mathsf{then}\ [\tau_1,\tau_2]A_1\ \mathsf{else}\ [\tau_3,\tau_4]A_2) \stackrel{\text{def}}{=}\\
&\quad\langle(\mathbf{if}\ [\tau_c]F\ \mathbf{then}\ \mathit{TransTST}([\tau_1,\tau_2]A_1)\\
&\qquad \mathbf{else}\ \mathit{TransTST}([\tau_3,\tau_4]A_2)),\\
&\qquad \tau \le \tau_c \le \tau' \wedge \tau_c \le \tau_1 \le \tau_2 \le \tau' \wedge \tau_c \le \tau_3 \le \tau_4 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',\mathsf{while}\ [t_c]F\ \mathsf{do}\ [t_1,t_2]A) \stackrel{\text{def}}{=}\\
&\quad\langle \mathbf{while}\ [t_c]\ F\ \ \mathit{TransTST}([t_1,t_2]A), \tau \le t_c \le t_1 \le t_2 \le \tau'\rangle\\
&\mathit{TransTask}(\tau,\tau',\mathsf{foreach}\ \bar{x}\ \mathsf{where}\ [\tau_c]F(\bar{x})\ \mathsf{do\ conc}\ [\tau_1,\tau_2]A(\bar{x})) \stackrel{\text{def}}{=}\\
&\quad\langle \mathbf{foreach}\ \bar{x}\ \mathbf{where}\ [\tau_c]F(\bar{x})\ \mathbf{do\ conc}\ \mathit{TransTST}([\tau_1,\tau_2]A(\bar{x})),\\
&\qquad \tau \le \tau_c \le \tau_1 \le \tau_2 \le \tau'\rangle
\end{aligned}$$

TransTST() should translate a full TALF C-ACT into a TST node. A composite action type specification is therefore translated as follows, given that $\mathit{TransTask}(\tau,\tau',T) = \langle T',\psi\rangle$:

$$\begin{aligned}
&\mathit{TransTST}([t,t']\underline{\mathsf{comp}}(\bar{v}) \rightsquigarrow [t,t']\mathsf{with}\ \bar{x}\ \mathsf{do}\ T\ \mathsf{where}\ \phi) \stackrel{\text{def}}{=}\\
&\quad \underline{\mathsf{comp}}(t,t',\bar{v}) = \mathbf{with}\ \bar{x}\ \ T'\ \ \mathbf{where}\ \phi\ \mathbf{and}\ \psi
\end{aligned}$$

Finally, a C-ACT can also occur unnamed and nested inside another composite action. Since all TST nodes must be named, a new name must be automatically generated. Then,

$$\begin{aligned}
&\mathit{TransTST}([\tau,\tau']\mathsf{with}\ \bar{x}\ \mathsf{do}\ T\ \mathsf{where}\ \phi) \stackrel{\text{def}}{=}\\
&\quad \mathsf{name}(\tau,\tau') = \mathbf{with}\ \bar{x}\ \ T'\ \ \mathbf{where}\ \phi\ \mathbf{and}\ \psi
\end{aligned}$$

Assuming that GOAL nodes have already been expanded through planning (Section 5), the translation from a TST back to a TALF composite action is very similar but slightly simpler, as no new names need to be generated. This bidirectional translation provides a formal connection between the two mission representations and a formal semantics for executable TSTs.

To explore this connection further, note that any composite action in TALF, together with the additional axioms and the circumscription policy discussed earlier, is associated with a set of logical models. Since TALF is a temporal logic, each model represents one possible evolution of the state of the world over time under the assumption that the composite action is executed. Additional information such as what is known about the current state of the world can also be added. Assuming that the information provided in the specification is correct, the actual trace that results from the execution of the corresponding TST in a robotic system must correspond directly to a member of that set of models. This permits formally grounded conclusions to be drawn about what will happen if a particular TST is executed.

4.3. *Semantics: Connecting TSTs to Delegation*

In Section 2.1 we defined the high-level semantics of the S-Delegate speech act, but the alternative pragmatic specification of the $\mathit{Can}_X(\tau)$ predicate had to wait until we had a concrete task structure in which it could be grounded. Now that this structure has been clearly specified, we can provide the missing information, which is also essential for the concrete realization of a delegation process in Section 6.

We first note that if an agent accepts the delegation of a task, it takes on the responsibility for ensuring that the entire task will be executed in a way that satisfies all associated constraints. Given the use of task specification trees, a task can be composite. Taking the responsibility for such a task should not necessarily mean that an agent must execute all elementary actions in the tree by itself or even that it will centrally coordinate the execution of the tree. However, if it does enlist the help of other agents, it must verify that those agents will accept the delegation of parts of the TST *before* it can accept the responsibility for the entire TST. For task specification trees, delegation will therefore be a recursive process.

This leads directly to the question of exactly how much of the task the responsible agent must actually perform itself. Given a tree-based representation, there is a natural answer: The agent that is delegated a TST must itself execute (run the executor for) the root node. Immediately re-delegating the *entire* tree to another agent would merely introduce an unnecessary middleman, in which case it would be better to backtrack and delegate to the other agent directly. This requirement is formally expressed in the definition of the *Can*() predicate for structural nodes [7]. On the other hand, we can permit any subtree rooted in a child node to be recursively delegated, to the same agent or another. An exception to the prohibition on re-delegation of the root node is naturally made at the top level of a mission – otherwise no agent (including an agentified graphical interface on a ground station) could ever initiate a mission whose root node must be performed by another agent.

The *intuitive* meaning of $\mathit{Can}_X(\tau)$ for a TST τ is thus that to the best of its knowledge, agent X is able to execute the root node of τ and ensure that all child nodes are executed, either by executing them itself or by delegating them to others. For instance, an agent *Can* perform the TST corresponding to "**with** $v_1, v_2, \ldots, v_n$ **sequence** $\alpha_1, \alpha_2, \ldots, \alpha_m$ **where** *cons*" if it can both

execute the top-level sequence node (which entails coordinating the execution of the child nodes in time) and ensure that all child nodes (the immediate subtasks) are executed in the given order within the execution interval permitted for the sequence.

Somewhat more formally, we assume that all agents can execute structural nodes such as sequences, as this is a standardized task that does not require specific hardware and that permits a standardized definition of $Can()$ that can be built into the architecture. For example, the definition of $Can()$ for a sequence essentially states that for each child node α_i, either the same agent $Can()$ execute α_i or there is another agent for which *Delegate*() succeeds. In the latter case we have recursed back into the semantics of S-Delegate, a recursion that ends at every elementary action node.

The definition of $Can()$ must be explicitly specified for each elementary action *type* and can be specified separately for each platform. For example, platform X can specify its own definition of $Can_X(\textbf{fly-to}(\ldots))$. It can then associate the definition with its own constraints on flying, such as its maximum velocity. Pragmatically, constraints can also be connected to functionalities such as motion planners to determine flight times, using semantic attachment [36] to augment the formal semantics used.

In addition to this, the set of constraints involved in each node must be logically consistent (there must be at least one assignment of values to variables that satisfies all constraints). The complete formal semantics of Can_X [7] specifies that constraints are not only checked in each individual node but are accumulated recursively, thereby verifying that the TST in question is globally consistent. In the implementation, the time, space and resource constraints involved can be modeled as a distributed constraint satisfaction problem (DisCSP). This allows us to apply existing DisCSP solvers to check the consistency of partial task assignments in the delegation process and to formally ground the process. Consequently, the *Can* predicate used in the precondition to the S-Delegate speech act is not only formally but also pragmatically grounded in the implementation.

5. Automated Planning for Mission Specifications

Given a specification of the current state of the world together with a declarative specification of the prerequisites and consequences of the available actions, a *task planner* can automatically search for a way of combining actions into a *plan* that achieves a given goal. For unmanned aircraft, actions may for example include taking off and landing, flying between waypoints or to a distant destination, hovering, positioning cameras, taking pictures, and loading or unloading cargo. Goals will vary depending on the nature of a mission and could involve having acquired pictures of certain buildings or having delivered crates of emergency supplies to certain locations after a natural disaster. The final plan can then be viewed as a mission specification consisting of a single composite action.

In the context of this article, task planners can be used to generate mission specifications in two distinct ways. First, one can specify a mission goal to be planned for (perhaps using a graphical interface), use the planner to generate a plan for the goal, and convert the plan into a fully specified TST that should then be delegated and executed (Figure 13). This is the approach that will be discussed in the current section. Second, one can use a more integrated approach where goal nodes are integrated into TSTs, with plan generation taking place incrementally during delegation. This is discussed in Section 7, after we have detailed the concrete delegation process as applied to TSTs.

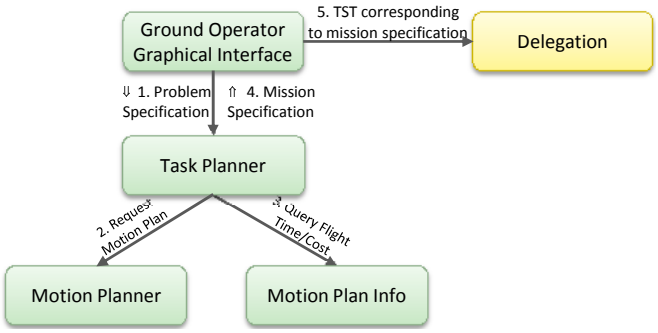

Figure 13. Planning before delegation.

5.1. *The TFPOP Planner*

Previously we have used TALplanner [37, 38] as a basis for the task planning capability in the UASTech system architecture. However, while TALplanner is both expressive and efficient, its plan structure is based on timed schedules. In a multi-agent setting this can sometimes lead to unnecessarily strict constraints on execution order. More recently we have developed a new planner, TFPOP [25,26,39], which has to a great extent replaced TALplanner in the architecture. The focus in developing TFPOP has been on allowing loose commitment to the precedence between actions while retaining from TALplanner the high level of performance generally associated with sequential planning.

Though flexible execution schedules can be achieved by using partially ordered plans instead of timed schedules, partial-order planners tend to be considerably slower than sequential or timed forward-chaining planners. The main reason for this is that traditional partial-order planning algorithms insert actions at arbitrary and varying points in a plan. This in itself improves the flexibility of the plan construction procedure, which is beneficial for performance. On the other hand, it only allows very weak information to be inferred about the state that will hold after any given action, which has turned out to be problematic. Forward-chaining planners, in contrast, can generate *complete* state information and take advantage of powerful state-based heuristics or domain-specific control [37, 40]. This leads to the question of whether some aspects of forward-chaining could be used in the generation of partial-order plans, combining the flexibility of one with the performance of the other – for example through the generation of stronger state information for partial-order plans. TFPOP is exactly such a combination: A sound and complete hybrid of temporal partial order causal link (POCL[d]) planning,

[d]We refer to Weld [41] for an overview of concepts and terminology associated with POCL planning.

generating plan structures suitable for concurrent execution, and forward-chaining, for efficiency.

As in standard partial-order planning, TFPOP retains a partially ordered plan structure at all times and provides a great deal of flexibility in terms of where new actions can be introduced during search. Creating highly informative states from a partial-order plan does require *some* form of additional structural constraint, though. The key is that for flexible execution in an unmanned system consisting of multiple agents, partial ordering is considerably more important between different agents than within the subplan assigned to any given agent: Each UAV can only perform its flight and delivery actions in sequence, as it cannot be in several places at once. Generating the actions for each agent in sequential temporal order is a comparatively small sacrifice, allowing actions belonging to distinct agents to remain independent to exactly the same extent as in a standard partial-order plan.

Each agent-specific *thread* of actions can then be used to generate informative agent-specific states in a forward-chaining manner, resulting in a Threaded Forward-chaining Partial Order Planner, or TFPOP for short. In particular, many state variables are associated with a specific agent and are only affected by the agent itself. This holds for the location of an agent (unless some agents can move others) and for the fact that an agent is carrying a particular object (unless one agent can "make" another carry something). Complete information about such state variables can easily be generated at any point along an agent's sequential thread and is in fact particularly useful when considering potential actions for the agent itself. For example, whether a given UAV can fly to a particular location depends on its own current location and fuel level, not those of other agents. Additionally, we have complete information about static facts. Taken together this usually results in sufficient information to quickly determine whether a given precondition holds, thus allowing TFPOP to very efficiently rule out most inapplicable actions. A fallback procedure takes care of the comparatively rare cases where states are too weak.

Planning Problems and Expressivity. The input language to TFPOP is a restricted form of TAL that can be concisely characterized in planning terms as follows.

Assume a typed finite-domain or *fluent* (state-variable) representation, where $\mathsf{loc}(crate)$ could be a location-valued fluent taking a crate of emergency supplies as its only parameter.

An *operator* has a list of typed parameters, the first of which always specifies the executing *agent*. For example, flying between two locations may be modeled using the operator $\mathsf{fly}(uav, from, to)$. Each operator o has a *precondition* formula $pre(o)$ that may be disjunctive and quantified. We currently assume all *effects* are conjunctive and unconditional and take place in a single effect state at the end of an action. An *action* is a fully instantiated (grounded) operator. This corresponds directly to a subset of the full expressivity of TAL actions.

Action *durations* often depend on circumstances outside the control of a planner or execution mechanism: The time required for flying between two locations may depend on wind, temporary congestion at a supply depot where it wants to land, and other factors. We therefore model *expected* action durations $expdur(a) > 0$ but never use such durations to infer that two actions will end in a specific order. Each duration is specified as a temporal expression that may depend on operator arguments.

Many domains involve mutual exclusion, as when only one UAV can refuel at a time. To explicitly model this, each operator is associated with a set of binary parameterized *semaphores* that are acquired throughout the execution of an action. For example, $\mathsf{use\text{-}of}(location)$ may be a semaphore ensuring exclusive access to a particular location. The set of semaphores acquired by an action a is denoted by $sem(a)$.

For any problem instance, the current version of TFPOP requires an *initial state* that completely defines the values of all fluents at time 0. Again, this uses a subset of the expressivity of TAL, where incomplete information is permitted. *Goal formulas*, like preconditions, may be disjunctive and quantified.

Plan Structure. A *plan* is a tuple $\pi = \langle A, L, O, M \rangle$, where:

(1) A is the set of all *action instances* occurring in the plan.
(2) L is a set of ground *causal links* $a_i \xrightarrow{\mathsf{f}=v} a_j$ representing the commitment that a_i will achieve the condition $\mathsf{f} = v$ for a_j. Causal links are a standard feature of POCL planning and are used to determine when an action might interfere with achievement relations between two other actions, in which case this interference must be corrected through new ordering constraints.
(3) O is a set of *ordering constraints* on A whose transitive closure is a partial order denoted by $\preceq_O$. The index of $\preceq$ will be omitted when obvious from context. As usual, $a_i \prec a_j$ iff $a_i \preceq a_j$ and $a_i \neq a_j$. The fact that $a \prec b$ means that a ends strictly before b begins. In TAL, such ordering constraints can be represented by temporal constraints related to the start and end timepoints specified for each action.
(4) M is a set of *mutex sets*, where no two actions belonging to the same mutex set can be executed concurrently (maybe due to explicit semaphores or because their effects interfere) but whose order is not necessarily defined before execution. This enhances execution flexibility compared to a pure partial order where such actions must be ordered at plan time. $M(a)$ denotes the possibly empty set of actions with which a is mutex according to M. In TAL, mutex sets can be represented by *disjunctive* temporal constraints related to the start and end timepoints specified for each action in the set.

Let $act(\pi, t)$ denotes the set of action instances executed by a given thread t. We add the *structural constraint* that such sets must be totally ordered by O.

Though we may use expected durations to infer approximately when an action is *expected* to start, actual durations may differ. Therefore expected durations cannot be used to infer new ordering constraints in O or to schedule actions to execute at specific timepoints. Instead, actions must only be invoked when notified that all predecessors have finished.

Note that plans with this structure are always dynamically controllable [42]. Note also that a plan can be directly translated into a TAL narrative or, as will be exemplified below, a TALF composite action (from which a TST can be generated).

Executable Plans. To determine whether a plan $\langle A, L, O, M \rangle$

is executable, we must consider all possible orders in which it permits actions to start and end. During the planning process each action $a \in A$ is therefore associated with an *invocation node* $inv(a)$ where preconditions must hold and semaphores are acquired, and an *effect node* $eff(a)$ where effects are guaranteed to have taken place and semaphores are released. Action precedence carries over directly to such nodes: If $a \in A$ then $inv(a) \prec eff(a)$, and if $a \prec b$ then $eff(a) \prec inv(b)$.

The *node sequences* that may arise from execution are exactly those that are consistent with $\prec$ and ensure that no two actions that are mutually exclusive according to M are executed concurrently. A plan is *executable* iff all associated node sequences $[n_0, n_1, \ldots, n_k]$ satisfy the following constraints. First, for every pair of distinct actions $\{a_i, a_j\} \subseteq A$ whose execution overlaps in the given node sequence:

(1) $sem(a_i) \cap sem(a_j) = \varnothing$: Two concurrently executing actions cannot require the same semaphore.
(2) $affected(a_i) \cap affected(a_j) = \varnothing$, where $affected(a)$ is the set of fluents occurring in the effects of a: Two concurrently executing actions cannot affect the same fluent.
(3) $affected(a_i) \cap required(a_j) = \varnothing$, where $required(a)$ is the set of fluents represented in incoming causal links to a: No action can affect a fluent used to support the precondition of another concurrently executing action.

Second, the effects of any action $a \in A$ must be internally consistent. Third, all preconditions must be satisfied: Let s_0 be the initial state and for all $0 < i \leq k$, let s_{i+1} be

$$\begin{cases} s_i \text{ modified with the effects of } n_i & \text{if } n_i \text{ is an effect node,} \\ s_i & \text{otherwise.} \end{cases}$$

Then for every invocation node n_i corresponding to an action a, we must have $s_i \models pre(a)$.

Solutions. An executable plan is a *solution* iff every associated node sequence results in a state s satisfying the goal.

5.2. *The TFPOP Planning Algorithm: Overview*

We now turn our attention to the problem of finding a solution, starting with an overview of the TFPOP search space, its use of partial states and the planner itself. A complete and detailed description is available in Kvarnström [26].

Search Space. The *initial node* in the search space is the executable plan $\langle \{a_0\}, \varnothing, \varnothing, \varnothing \rangle$, where a_0 is a special initial action not associated with a specific thread. This action has no preconditions or semaphores and its effects define the initial state, as is common in POCL planning. For all other $a_i \in A$ in any plan, $a_0 \prec a_i$.

A *successor* is an executable plan π where exactly one new action has been added to the end of a specific thread t (occurring last in $act(\pi, t)$), possibly also constrained to occur before or after certain actions in other threads.

Partial States. Our intention is to improve performance by generating informative states to be used in precondition evaluation. Each action in a plan is therefore associated with a state specifying facts known to be true from the instant the action ends until the start of the next action in the same thread, or "forever" if there is no next action yet. The initial action is a special case, belonging to no thread and providing an initial state for *all* threads.

In general, one cannot infer complete information about any point during the execution of a partially ordered plan. Therefore, a formula can evaluate to *true*, *false*, or *unknown*.

Also, ensuring states contain all information that *can* theoretically be inferred from a partial order requires comparatively complex state structures and state update procedures. This tends to require more time than is saved by rapid evaluation. Formula evaluation is therefore permitted to return *unknown* even when it would be theoretically possible to determine whether a formula holds at a certain point in a plan. The complete fallback procedure *make-true* is then used to explicitly search the plan structure for support [26].

As maximal state information is not required, a *partial state* can be a simple structure specifying a finite set of possible values for each fluent ($f \in \{v_1, \ldots, v_n\}$). The partial evaluation procedure $eval(\phi, s)$ is the natural extension of a standard recursive formula evaluator to three truth values. For example, suppose ϕ is $\alpha \wedge \beta$. If $eval(\alpha, s) = false$ or $eval(\beta, s) = false$, then $eval(\phi, s) = false$. If both subformulas are *true*, the conjunction is *true*. Otherwise, the conjunction is *unknown*. This combination has proven to yield a good balance between information and speed, representing most of the useful information that can be inferred but also supporting very efficient state updates and formula evaluation.

The Planner. We will now outline the operation of the TFPOP planner at a high level of abstraction. Note that **choose** refers to standard non-deterministic choice, in practice implemented by iteration over possible choices with backtracking.

procedure TFPOP
$\pi \leftarrow \langle \{a_0\}, \varnothing, \varnothing, \varnothing \rangle$ *// Initial plan*
repeat
  if goal satisfied **return** π
  choose a thread t to which an action should be added
  // Use partial state to filter out most potential actions
  $s \leftarrow$ partial state at the end of thread t
  choose an action a for t such that $eval(pre(a)) \neq false$
  if effects of a are inconsistent **then fail** (backtrack)
  // Complete check:
  // Can the action really be added, and how?
  choose precedence constraints C and causal links L
    using *make-true*(), ensuring $pre(a)$ is satisfied,
    a does not interfere with existing actions
    and no existing action interferes with a
  update resource usage
  if resource constraints violated **then fail** (backtrack)
  add a, C, L and necessary mutex sets to π
  update existing partial states
  create new partial state for a

First, the planner tests whether the goal is already satisfied, which could be the case even for the empty plan if the goal is trivial. The goal is satisfied iff it holds after all actions in the plan have finished, which can often be determined by conjoining information from the final states of all threads but sometimes

requires a more complex evaluation procedure. If the goal is satisfied, a solution is returned. If not, a successor must be found.

By definition, any successor of a node adds a single new action a to the end of an existing thread t. The planner therefore begins by determining which thread t it should first attempt to extend. A variety of policies can be used here, where the default one aims to minimize makespan by distributing actions as evenly as possible across available threads. It therefore prefers to extend threads whose current actions are expected to finish earlier (*calculate-expected-times*). The last state associated with the thread to be extended specifies facts that must hold when a is invoked and can therefore be used to evaluate the preconditions of potential candidates.

The result may be *false*, in which case a candidate action has efficiently been filtered out. Otherwise, the result must have been either *true* or *unknown*. Regardless, we verify that the action's effects are consistent. If they are, we call *make-true* to search for precedence constraints and causal links that can be added to ensure the precondition will hold (which might fail if the precondition was *unknown*) and that there is no interference between a and existing actions in the plan (*fix-interference*).

Resource usage is updated and resource constraints are checked. If no constraints are violated, we determine which existing actions $a' \in A$ share a semaphore or an affected fluent with the new action a and update the set M of mutex sets to ensure such actions cannot be executed in parallel with a. As this cannot fail, a new successor has been found.

The existing partial states represent facts that hold during a certain interval of time. If the effects of the new action may occur within that interval, the state must be updated and "weakened" to remain sound. Suppose for example that the state s associated with $a_1 = \mathsf{fly(uav8,depot1,loc12)}$ claims that $\mathsf{loc(uav5)} \in \{\mathsf{depot4}\}$: Given the current plan, this fact *must* hold from the end of a_1. Suppose further that $a = \mathsf{fly(uav5,depot4,loc57)}$ is added, and that its effects may occur within the interval of time where s should be valid. Then, s must be updated to claim that throughout its interval, $\mathsf{loc(uav5)}$ takes on one of two possible values: $\mathsf{loc(uav5)} \in \{\mathsf{depot4}, \mathsf{loc57}\}$.

A new partial state must be generated that is valid from the end of the new action until infinity. To do this we first copy the state of the action's predecessor in the same thread, which must be valid when the action is invoked. We then strengthen the state using information gathered from the causal links generated by *make-true*. We also apply all effects of the new action, resulting in a state valid exactly when a ends. Finally, we weaken this state using all effects that may possibly occur in other threads from a until infinity, in a procedure essentially identical to the state update above. The result is a new sound state for a and a new executable plan, and the procedure can repeat.

Search Guidance and Performance. TFPOP is currently not guided by heuristics but by precondition control formulas [43]. Such formulas represent conditions that are required not for executability but for an action to be meaningful or useful in a given context, and have been used to great effect in TLPLAN [40] and TALplanner [37].

Control formulas often need to refer to the goal of the current problem instance. For example, this is needed to determine where the crates on a carrier should be delivered, so UAVs can be prevented from flying the carrier to other locations – which would be possible but pointless. TFPOP therefore supports the construct $\mathsf{goal}(\phi)$ [40], which is used in preconditions to test whether ϕ is entailed by the goal.

Though any planner supports preconditions, most partially ordered planners use means-ends techniques where stronger preconditions may generate new subgoals to resolve but provide no guidance. TFPOP requires immediate support for all preconditions within the current plan and has efficient precondition evaluation procedures, yielding both efficient and effective pruning of the search space. Empirical evaluation shows the resulting knowledge-rich planner to be very close to TLPLAN and TALplanner in performance, while using a considerably more expressive and flexible plan structure [26].

5.3. *The TFPOP Planning Algorithm: Details*

We will now discuss certain aspects of TFPOP in more detail.

Selecting a Thread to Extend. As stated above, the planner aims to minimize makespan by distributing actions evenly across available threads. It therefore prefers to extend threads whose current actions are expected to finish earlier given the assumption that each action will start as soon as its explicit predecessors have finished and it can acquire the required semaphores.

This prioritization can be done by calculating the expected start time $expstart(a)$ and finish time $expfin(a)$ for every action $a \in A$ using the procedure below. The implementation uses a variation where times are updated incrementally as new actions are added. Again, timing cannot be used to infer that one action *must* occur before another.

procedure *calculate-expected-times*$(\pi = \langle A, L, O, M \rangle)$
$expstart(a_0) = expfin(a_0) = 0$ *// Initial action a_0 at time 0*
while some action in A remains unhandled
  $E \leftarrow \{a \in A \mid$ all parents of a are handled$\}$ *// Executable now*
  // We know $E \neq \emptyset$, since the partial order is non-cyclic.
  // Calculate for each $a \in E$ when its parents should finish.
  // If several actions could start first, break ties arbitrarily.
  forall $a \in E$: $t(a) = \max_{p \in parents(e)} expfin(p)$
  $a \leftarrow$ arbitrary action in E minimizing $t(a)$ *// Could start first*
  $expstart(a) \leftarrow t(a); expfin(a) \leftarrow expstart(a) + expdur(a)$
  forall unhandled $a' \in M(a)$: $O \leftarrow O \cup \{a \prec a'\}$

The final step temporarily modifies O to ensure that when one action acquires a particular mutually exclusive resource, it will strictly precede all other actions that require it but have not yet acquired it.

Satisfying Preconditions. When a thread t has been selected, each potential action a for the thread is considered in turn. For a to be applicable, it must first be possible to satisfy its preconditions.

Let a_L be the last action currently in the thread t (with $a_L = a_0$ if t is empty), and let s be the last state in t. If $eval(pre(a), s) = false$, then a cannot be applicable and we can immediately continue to consider the next action.

If *unknown* is returned, the reason may be that $pre(a)$ requires support from an effect that (given the current precedence

constraints) may or may not occur before a, that there is potential interference from other actions, or simply that the state was too weak to determine whether $pre(a)$ holds. As we do not know, we must test whether we can introduce new precedence constraints that ensure $pre(a)$ holds. Even if the precondition was *true*, guaranteeing that support can be found, we must still generate new causal links to protect the precondition from interference by actions added later.

A provisional plan $\pi' = \langle A \cup \{a\}, L, O \cup \{a_L \prec a\}, M\rangle$ is created, where a is placed at the end of its thread. Then, *make-true*(ϕ, a, s, π') determines whether it is possible to guarantee that $\phi = pre(a)$ holds when a is invoked in π' in the partial state s. A set of *extended plans* is returned, each extending π' with links and possibly constraints corresponding to one particular way of ensuring that ϕ will hold. If this is impossible, the empty set of extensions is returned.

procedure *make-true*$(\phi, a, s, \pi = \langle A, L, O, M\rangle)$
if ϕ is $v_1 = v_2$
  if $v_1 = v_2$ **return** $\{\pi\}$ **else return** $\varnothing$
else if ϕ is goal(ϕ)
  if goal state $\models \phi$ **return** $\{\pi\}$ **else return** $\varnothing$
else if ϕ is $\mathsf{f} = v$
  if $s \models \mathsf{f} \neq v$ **return** $\varnothing$ *// Use state for efficiency*
  // Find potential supporters a_i that may be placed before a
  $X \leftarrow \varnothing$
  forall $a_i \in A$ **such that** a_i assigns $\mathsf{f} = v$ and not $a \prec a_i$
    // Add supporter, find ways of fixing interference (below)
    $\pi' \leftarrow \langle A, L \cup \{a_i \xrightarrow{\mathsf{f}=v} a\}, O \cup \{a_i \prec a\}, M\rangle$
    $X \leftarrow X \cup$ *fix-interference*$(\mathsf{f}, a_i, a, \pi')$
  return X
else if ϕ is $\neg(\mathsf{f} = v)$ *// Handled similarly*
else if ϕ is $\neg\alpha$
  Push negation inwards using standard equivalences such as $\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$ *and recurse*
else if ϕ is $\alpha \wedge \beta$
  // For each way of satisfying α, find all ways of also satisf. β
  $X \leftarrow \varnothing$
  forall $\pi' \in$ *make-true*(α, a, s, π)
    $X \leftarrow X \cup$ *make-true*(β, a, s, π')
  return X
else if ϕ is $\alpha \vee \beta$
  // Find all ways of satisfying either α or β
  return *make-true*$(\alpha, a, s, \pi) \cup$ *make-true*(β, a, s, π)
else if ϕ is quantified
  Instantiate with all values of the variable's finite domain and handle as a conjunction or disjunction
end if

Avoiding Interference. As shown above, when *make-true* finds a possible supporter a_i for an atomic formula $\mathsf{f} = v$, it is placed before a and a causal link is added. The *fix-interference* procedure then determines if and how we can guarantee that no other actions can interfere by also assigning a value to f between the end of a_i and the start of a. It iterates over all potential interferers a', adding (if permitted by π) constraints that place a' either before a_i or after a. After each iteration, X contains all minimally constraining ways of avoiding interference from all actions considered so far (and possibly some that are more constraining than necessary, which could be filtered out).

procedure *fix-interference*$(\mathsf{f}, a_i, a, \pi = \langle A, L, O, M\rangle)$
$X \leftarrow \{\pi\}$
forall $a' \in A \setminus \{a, a_i\}$ assigning a value to f
  if $a' \prec_O a_i$ or $a \prec_O a'$ **then** a_i cannot interfere **else**
    $X' \leftarrow \varnothing$ *// Then, for every solution to the earlier interferers*
    forall $\pi' = \langle A', L', O', M'\rangle \in X$
      // Can we place a' before the supporter a_i?
      if not $a_i \prec_{O'} a'$ **then**
        $X' \leftarrow X' \cup \{\langle A', L', O' \cup \{a' \prec a_i\}, M'\rangle\}$
      // Can we place a' after a?
      if not $a' \prec_{O'} a$ **then**
        $X' \leftarrow X' \cup \{\langle A', L', O' \cup \{a \prec a'\}, M'\rangle\}$
      // Note that both extensions may be impossible,
      // leading to a reduction of the size of X!
    $X \leftarrow X'$
return X

This ensures no actions in π can interfere with the preconditions of a, but a may also interfere with existing actions in π. That is handled by identifying all actions $a' \in A$ whose incoming causal links depend on fluents f affected by a and then preventing interference using *fix-interference*.

5.4. *A Logistics Example*

Consider a mission whose objective is to assist emergency services personnel in delivering food, water and medical supplies after a natural disaster where road systems have become inaccessible, such as the tsunami disaster in Japan.

To simplify the presentation, we assume a number of UAVs have already cooperated in scanning a large geographic area and have detected several injured people in need of supplies. The scanning functionality is implemented by searching for salient features in fused video streams from color and thermal cameras [3] and results in a saliency map pinpointing potential victims and their geographical coordinates, associating them with sensor input such as high resolution photos and thermal images. This information has been shown to a ground operator, who has used an interface specific to this mission type in order to approve a number of locations where certain amounts of supplies should be dropped off. Since the operator does not want to manually specify all details, the next step is then to automatically generate a plan that efficiently delivers the required supplies using the resources at hand.

Figure 14. A supply depot (left) and a carrier (right) in simulation.

Specifically, there is at least one *supply depot* (Figure 14) containing a known number of crates of each type (food, medicine, water). There are a number of smaller UAVs that can carry individual crates. A number of larger UAVs can carry *carriers* (Figure 14) containing several crates, which is considerably more efficient over longer distances. Smaller UAVs can cooperate to place crates on carriers for larger UAVs to deliver.
The mission-specific interface can use this information to generate a planning problem instance providing relevant information about the current state of the world, such as where depots are located, as well as the goals of the mission. The interface can then directly call the planner, which generates a plan in the form of a mission specification.

The following is a small part of the composite action output generated by TFPOP for an example problem within the given context. Each thread is by default independent of the others, resulting in a parallel composition of threads. Within each thread actions are sequential. We see that UAVs *small3* and *small6* are both involved in loading crates onto *carrier0* and *carrier2*. UAV *heli1* will deliver the carrier, but according to the temporal constraints encoding a precedence ordering between distinct threads, this cannot start before the corresponding **load-carrier** actions have completed. This output can be converted directly to a task specification tree that can be delegated to a group of unmanned aircraft.

with $start$, end, end_{11}, end_{13}, end_{15}, end_{17}, end_1, end_5, end_7, end_9, $start_{19}$, ... do (
  with $tstart_0$, $tend_0$ do [$tstart_0$,$tend_0$] (
    [$start_{25}$,end_{25}] **load-carrier**(small3,carrier0,d0,p0,medicine);
    [$start_{65}$,end_{65}] **load-carrier**(small3,carrier2,d2,p2,food)
)
||
  with $tstart_2$, $tend_2$ do [$tstart_2$,$tend_2$] (
    [$start_{23}$,end_{23}] **load-carrier**(small2,carrier0,d0,p0,medicine);
    [$start_{63}$,end_{63}] **load-carrier**(small2,carrier2,d2,p2,food)
)
||
  with $tstart_3$, $tend_3$ do [$tstart_3$,$tend_3$] (
    [$start_{47}$,end_{47}] **deliver-carrier**(heli1,carrier0,d1,depot0,p1);
    [$start_{49}$,end_{49}] **deliver-box-from-carrier**(heli1,carrier0,p1,medicine);
    [$start_{51}$,end_{51}] **deliver-box-from-carrier**(heli1,carrier0,p1,medicine);
    [$start_{53}$,end_{53}] **deliver-box-from-carrier**(heli1,carrier0,p1,medicine);
    [$start_{55}$,end_{55}] **deliver-box-from-carrier**(heli1,carrier0,p1,food);
    [$start_{57}$,end_{57}] **return-carrier**(heli1,carrier0,p1,d1)
)
||
  ...
) where
  $end_1 < start_{19} \wedge end_1 < start_{21} \wedge end_1 < start_{23} \wedge$
  $end_1 < start_{25} \wedge \ldots \wedge end_{25} < start_{47} \wedge end_{23} < start_{27} \wedge \ldots$

After discussing the delegation process in more detail, we will return to planning and show how it can be integrated with delegation in several different ways.

6. Delegation

As illustrated in Figure 4 on page 4, a ground operator can generate a formal mission specification in TALF in several ways, including through a graphical user interface adapted to a particular mission type and through the use of general automated planners such as TFPOP. Regardless of where a mission specification originates, we now know how to translate it into a task specification tree. We have also seen how to formally define the conditions required for such a tree to be delegated using the S-Delegate speech act to one or more agents having the abilities required for execution. The next step is to provide a realization of S-Delegate in terms of a concrete *delegation process* that can be implemented in a multi-agent system.

Though S-Delegate as a speech act is mainly concerned with agreements and commitment relative to two given agents, the delegation process defined below is also responsible for *task allocation*, which includes *finding* agents capable of performing and being responsible for parts of a complex mission. Integrating delegation and task allocation in an alternating, incremental and recursive process is considerably more efficient than allocating agents to a complete complex task at once, especially in case backtracking is necessary.

6.1. *Executability, Constraint Problems and Task Allocation*

To carry out the task allocation aspect of the delegation process, we require a means of determining whether a particular agent *can* execute a particular TST node. For this to be the case, the agent in question must have the required capabilities and resources. This is represented and verified using constraints linking the *required* capabilities and resources for each node to the executing agent's *available* capabilities and resources. If the constraint problem constructed in this way is consistent (has a solution), then this provides a strong guarantee about validity and quality which is necessary for safety and reliability.

The *available capabilities* of an agent and its platform are generally fixed and can therefore be described using constraint variables explicitly constrained to take on specific values. For example, some platforms have cameras and can take pictures, while others do not. The *available resources* will vary depending on the agent's existing commitments, including the tasks it has already been allocated. These commitments can also be represented in the constraint stores and schedulers of the agent in question, where each resource is modeled as a variable and each current commitment as a constraint related to that variable.

Constraints related to *required* capabilities and resources originate from two distinct sources.

First, each TST node provides a set of constraints originating in a where clause in a mission specification. These constraints represent requirements placed on mission execution by the delegator, and may ultimately have been defined by a ground operator. For example, in a certain mission, we may not be permitted to fly at an altitude above 30 meters, and we may not be permitted to intrude into a certain area during a certain period of time – a spatio-temporal constraint representing a no-fly-zone.

Second, each agent provides a specific set of constraints for each type of node that it can execute. These constraints represent mission-independent restrictions on how a particular agent and its platform is physically able to execute the given node type, and serve as a concrete realization of the conditions formally specified by the $Can_X()$ predicate. For example, flight-capable platforms of different types such as LinkQuads and RMAXes would have their own specific constraints on their maximum flight velocity and acceleration, which affects whether the aircraft can fly to a certain location within a temporal interval. Similarly, there could be constraints related to the maximum resolution of a picture that can be taken with the on-board camera and on how closely the camera can zoom in.

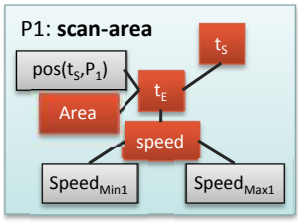

Figure 15. Agent-specific constraint structure for **scan-area**.

Figure 15 shows the constraint variables involved in such a "constraint template" for elementary action nodes of type **scan-area** for agent/platform P_1. Red/dark rectangles represent node parameters in the node interface, which must be present regardless of which agent is involved. Light gray rectangles are local variables associated with the agent's own constraint model for the **scan-area** action, where other agents could use other values for $\mathsf{Speed}_{\mathsf{Min1}}$ and $\mathsf{Speed}_{\mathsf{Max1}}$ but could also use completely different constraint networks. Edges represent the fact that variables are dependent, occurring in the same constraint.

The actual constraints involved in the network for **scan-area** may include $\mathsf{Speed}_{\mathsf{Min1}} \leq \mathsf{speed} \leq \mathsf{Speed}_{\mathsf{Max1}}$, defining a range of permitted speeds. Similarly, the constraints for a **fly-to** action could include $\mathsf{t}_E = \mathsf{t}_S + \mathit{distance}(\mathsf{from},\mathsf{to})/\mathsf{speed}$, defining the expected duration of the flight in terms of the formal node parameters. As stated earlier, the latter constraint could also be connected to a motion planner through semantic attachment [36] for increased precision in time estimates. Depending on the expressivity of the constraint solver being used, increasingly sophisticated constraints could be defined both by an agent, specifying the conditions under which it *can* execute certain actions, and in a mission specification, determining the conditions under which the ground operator *wants* the mission to be executed.

To determine whether a particular agent can achieve the task associated with a specific TST node, all of these constraints are instantiated and added to a constraint store. Agent-specific constraints are then connected to mission-specific constraints via the node parameters in the node interface. Taken together, this models the general and agent-specific capability and resource requirements for the node, as well as the availability of capabilities and resources. Figure 16 illustrates the constraint network when node N_2 from a small scanning mission has been allocated to agent P_1 and we are determining whether node N_3 can be allocated to agent P_2. Template variables such as speed have been instantiated with indexes corresponding to TST nodes (speed_2, speed_3) in order to avoid ambiguity, and are constrained to be equal to the corresponding actual node parameters (dashed lines). The constraint networks used for **scan-area** for these agents happen to be very similar but use distinct upper and lower bounds on speed.

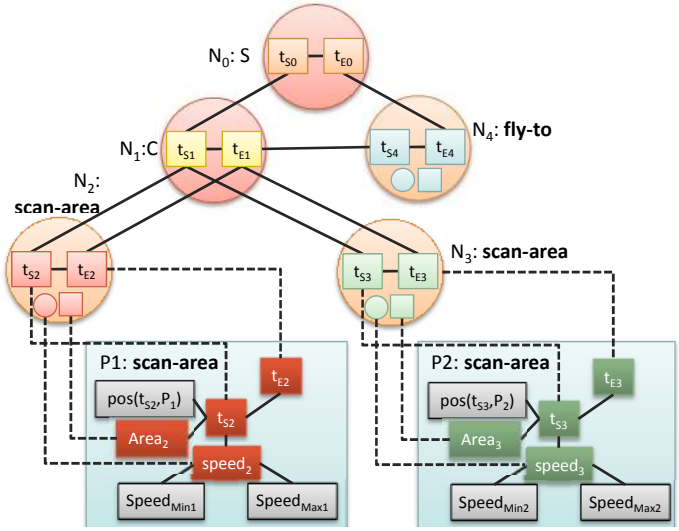

Figure 16. The combined constraint problem structure after allocating node N_2 to platform P_1 and node N_3 to platform P_2.

When the combined delegation and task allocation procedure detailed below proceeds recursively through a task specification tree, a constraint problem is incrementally constructed and extended each time a node is provisionally allocated to an agent. This yields a constraint problem representing all the constraints for this particular (partial) allocation of the TST, making constraint solving an integral part of the solution to the task allocation problem. If a consistent solution for the constraint problem is found then a *valid allocation* has been generated and verified, and the part of the mission that has been allocated is executable. The delegation procedure can then proceed to find a contractor for the next unallocated node in the tree and extend the constraint problem accordingly. If at any point the constraint problem becomes inconsistent, the current allocation is not valid and the procedure must backtrack. If all nodes have been successfully assigned, the assignment of timepoints to temporal parameters in each solution to the constraint problem can be seen as a potential execution schedule for the TST.

Since constraint variables are distributed among the agents, a *distributed* constraint satisfaction problem is generated. As each agent might control more than one resource with potentially complex constraints among them, distributed constraint reasoning with *complex local problems* is required [44]. Currently the constraint problem is represented using MiniZinc [45, 46]. This representation is translated into a solver-specific constraint language and solved. We currently use the Minion [47] constraint solver for local constraint problems, which is a

general finite domain constraint solver. For distributed constraint problems we have used our own implementation of the Asynchronous Weak Commitment Search (AWCS) algorithm [48].

The process of incrementally extending the constraint problem has a formal basis in a recursive procedure reducing the *Can* predicate statements associated with each task node with formally equivalent expressions, beginning with the top node, until the entire tree has been processed. The reduction of *Can* for an elementary action node contains no further *Can* predicates, since the ability to perform an elementary action only depends on the agent itself. Therefore the logical statements can be reduced to a constraint network, as exemplified in more detail in Doherty et al. [7].

In summary, a successful delegation and task allocation will yield a TST where all nodes are allocated to specific agents in a way that leaves all constraints satisfiable. During the process, a consistent network of distributed constraints is incrementally generated which guarantees the validity of the multi-agent solution to the original problem, provided that no additional contingencies unaccounted for in the mission specification or the definition of the platforms' capabilities arise when the TST is actually executed.

When verifying that the network is satisfiable, we may as a side effect also generate one specific solution to the constraint problem, where one specific value is assigned to each node parameter. However, this solution is not necessarily fixed: Unless specifically forbidden by the delegator, the contracting agents are allowed to find a new solution to the constraint problem even after delegation has finished, as long as the original mission-specific constraints defined by the delegator remain satisfied. This is important in case of contingencies: If one action takes longer than expected, we may still be able to satisfy the overall deadline of the mission by increasing flight velocity in the remainder of the mission, which requires finding a new constraint solution. It is also important if agents involved in one mission are also delegated parts of another mission, in which case modifying the execution parameters of the first mission (while still satisfying all of the delegator's constraints) may be required in order to accommodate the requirements of the second mission.

This approach grounds the original formal specification of S-Delegate with the actual processes used in the implementation, while retaining a formal definition in terms of a distributed constraint network which is in effect a formal representation.

6.2. *The Delegation Process*

We now turn our attention to describing the computational *delegation process* that is used to realize the S-Delegate speech act in a robotic platform. This process takes as input a complex task or mission represented as a TST and aims to find an appropriate agent or set of agents capable of achieving the mission through the use of recursive delegation. More specifically, its input is a reference to the root node of the TST, from which all other information can be gathered. If the allocation of agents in the TST satisfies the associated constraints and is approved by the delegators recursively, then the mission can be executed. Note that the mission schedule will be distributed among the agents involved and that depending on the temporal constraints used, the mission may not start immediately. However, commitments to the mission will have been made in the form of constraints in the constraint stores and schedulers of the individual platforms. Note also that the original TST given as input does not have to be completely specified but may contain goal nodes which require expansion of the TST with additional nodes.

We will discuss the delegation process on several levels, beginning with the initial definition of a mission.

6.2.1. *Specifying Missions and Initiating Delegation*

At the highest level, mission specification and delegation can be described using the following steps (Figure 17).

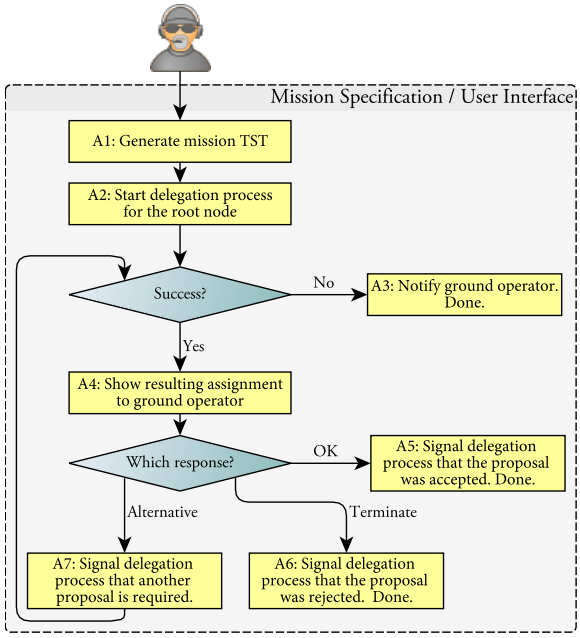

Figure 17. Delegation: Starting the delegation process.

Step A1: Generate a mission TST. A TST τ describing a particular mission is generated, for example through the use of a graphical interface on an agentified ground station which calls the agent's own TST factory to generate the tree.

Step A2: Request delegation. An agent A is requested to delegate the newly generated TST τ. This is normally the same agent where the TST was generated, in which case the request can be an internal service call as opposed to a speech act. The delegating agent does not necessarily have to be involved in actually executing the mission. Nevertheless, it does initiate the delegation process for τ, causing the entire mission to be recursively delegated to one or more executing agents. This process will be described in detail in the next subsection.

Step A3: Indicate failure. If delegation failed, for example due to lack of available resources or suitable platforms, this is indicated to the ground operator. As all possible alternatives have

already been tested, the delegation process has finished, though the operator may choose to modify the mission and try again, starting at step A1.

Step A4: Response from the ground operator. If delegation succeeded, the result is that an instantiated TST, with all nodes allocated to specific agents and with all constraints satisfiable, is proposed. At this point we have verified some of the preconditions of the S-Delegate speech act. For example, since the contractor B replied with a proposal we know that it believes it can do the task τ, and given that the delegator A trusts the contractor, A also believes that B can perform τ. This is discussed in more detail in step B1 below.

Many of the postconditions of S-Delegate are also *provisionally* represented in the constraint stores and schedulers of the agents involved. This is required in order to guarantee that the task remains feasible until the actual delegation takes place. Otherwise, there would be a race condition where a contractor could take on another task that may interfere.

The instantiated TST then has to be approved for execution. For example, the delegating agent can present the TST to the ground operator in the shape of a tree, a formal mission specification, or a graphical visualization adapted to the mission type. This step serves as a verification that the mission specification was correct and complete, so that the delegation process did not yield unintended results. Depending on the response, we proceed to step A5, A6 or A7.

Step A5: Accept the proposal. If the ground operator accepts the current proposal in step A4, the delegating agent signals this to the delegation process. This causes *accept-proposal* messages to be recursively sent to all contractors involved in the proposal (step B3i below), informing them that their proposals have been accepted. In a sense, this step constitutes the actual execution of S-Delegate and causes the postconditions of the speech act to occur definitely, not only provisionally.

Step A6: Terminate delegation. If the ground operator rejects the proposal and wants to terminate delegation of the mission as currently specified, the delegating agent signals this to the delegation process. This causes *reject-proposal* messages to be recursively sent to all contractors involved in the proposal (step B3j below). One effect of this is that all provisional constraints added during the process are retracted.

Step A7: Request an alternative proposal. If the ground operator rejects the current proposal but requests an alternative, the delegating agent signals this to the delegation process. The delegation process continues searching for solutions where it left off. If an alternative proposal is found, it may involve different allocations of tasks to agents and/or alternative expansions of goal nodes and other expandable nodes. The process again verifies success and returns to step A3 or A4.

Execution. Once an *accept-proposal* message has been received by a contractor for a particular node in the TST, the actual delegation process has finished. The contractor's execution subsystem is then permitted to start the executor for this node. This executor must ensure that all related timing and precedence constraints are satisfied before the actual execution begins. This may involve waiting for a specific timepoint or until certain other nodes have been executed.

While the mission is being executed, the contractors are committed to the constraints agreed upon during the approval of the tasks. These constraints should be monitored continually, which can be facilitated by an execution monitoring framework such as the one in Doherty, Kvarnström and Heintz [31]. Should constraints be violated, a repair process must be initiated. This process can then find alternative ways of achieving tasks.

For example, the contractors have a degree of autonomy during execution in the sense that they can modify internal parameters associated with tasks as long as they do not violate those constraints externally agreed upon in the delegation process. These parameters may be changed on the fly in order to avoid constraint violations. For example, if an aircraft is delayed, it may increase its target speed parameter for the remainder of its task as long as this does not violate speed constraints posed by the delegating agent.

Depending on the degree of autonomy permitted, more extensive repairs could be performed. For example, this might involve locally modifying a TST in order to achieve a simple or complex task using different elementary actions. Goal nodes could also be re-expanded given new information about the world gathered during execution, if that information contradicts what was predicted earlier. This would result in a new plan adapted to current circumstances. If such changes violate the mandated degree of autonomy, a contractor must receive confirmation from its delegator before proceeding with the modification.

6.2.2. *Initiating Delegation for the Root Node*

Recall that an agent that delegates the root node of a TST does not have to be actively involved in its execution. The delegation process for the root node is therefore somewhat different (and simpler) than delegation for inner nodes in a TST. We will therefore describe this aspect of delegation now, under the assumption that delegation is requested as described in step A2 above. See also Figure 18, where thick arrows indicate speech acts being sent or received.

Step B1: Find potential contractors. When the delegation capability is *asked to delegate* a task τ (step A2), both the delegating agent A and the task τ are fixed. It is also possible for the contractor B to be determined in advance through a constraint on the platform parameter of the root node of τ, for example by specifying a particular agent as the only possible contractor for the node or by specifying that its contractor must be the same as for another node that will be allocated to an agent earlier in the delegation process. In general this is not the case, which is why task allocation is integrated into allocation. Delegation must then *find* an agent B that satisfies the four preconditions of S-Delegate$(A, B, \tau = \langle \alpha, \phi, \mathit{constraints} \rangle)$ as described in Section 2.1.

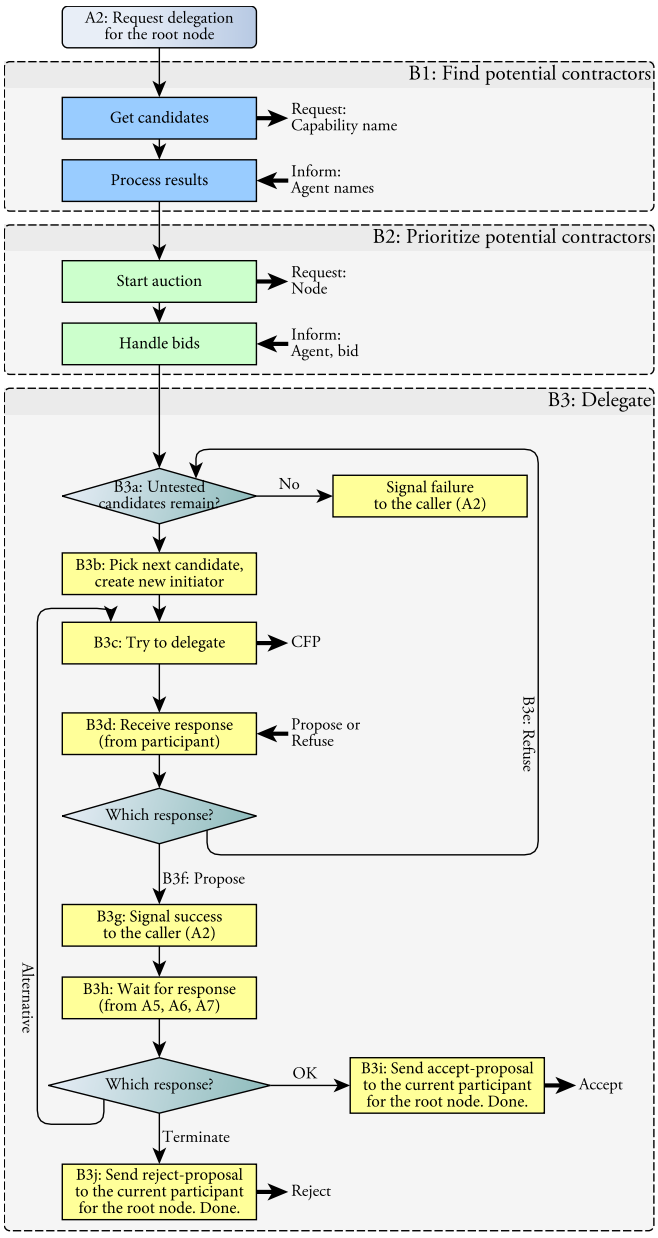

Figure 18. Delegation: Root node.

Precondition (1) was $Goal_A(\phi)$. We assume implicitly that this is satisfied, since an agent should never request the delegation of a task whose execution it does not desire.

Precondition (3) was $Bel_A(Dependent(A,B,\tau))$. To satisfy this, the delegation process can prioritize attempting to delegate tasks back to the delegating agent ($A = B$) in the next step (B2). If this turns out not to be possible, but another agent B is found that can perform the task, then A is dependent on B for τ.

Finally, preconditions (2) and (4) were that $Bel_A Can_B(\tau)$ and $Bel_B Can_B(\tau)$: Both the delegator and the contractor must believe that the contractor can perform the task. This could be verified in at least two ways. Agent A could have a knowledge base encoding complete knowledge about what all other agents can do, and what they *believe* they can do. This could be used to determine which agents satisfy the preconditions, but is problematic because it would be difficult to keep such a knowledge base up-to-date and it would be quite complex given the heterogeneous nature of the platforms involved. Additionally, the pool of platforms accessible for any given delegation at a given time is not known since platforms come and go. Instead, contractors will be found in a more distributed manner through communication among agents as a part of the delegation process.

To improve performance when the contractor B is not predetermined, we note that the definition of *Can()* entails that B will not be allowed to recursively delegate the root node n to someone else, even though it can recursively delegate n's children: This would leave B as an unnecessary middleman. This restriction allows the delegation process to filter out irrelevant agents at an early stage, as it can broadcast a query for agents capable of executing nodes with the same type as the root node n and limit delegation attempts to such agents. The ability to execute a node of a particular type is for the time being considered to be static and atemporal, which allows capabilities to be checked very rapidly. The delegator should thus attempt to delegate the task only to those potential contractors that have the necessary static capabilities. More complex or time-dependent abilities such as the ability to fly from one specific location to another at a given speed during a particular interval of time are then tested through general resource reasoning when a call for proposals is sent.

Step B2: Prioritize potential contractors. The result of the broadcast for capabilities contains no information about the usefulness or cost of allocating the node to particular candidate contractors. At the same time, blindly testing candidates for a node is an obvious source of inefficiency, even when some candidates have been filtered out. In the context of the recursive delegation process, this leads to a multi-robot task allocation (MRTA) problem, an important research issue in the multi-agent community [49–54].

This problem is currently approached by using a heuristic function based on *auctions* [55, 56] to determine the order in which platforms are tested. The delegator therefore broadcasts to the candidates a request for *bids* for the top node n. Each bid can reflect a potential contractor's estimated cost or suitability for accepting the task. To estimate the cost for a potential task, an agent might have to solve a constraint problem in order to find out the extra resources needed to take on the task relative to the current partial allocation. The cost could for example be related to the total time required to execute all tasks allocated to the platform. This increases the chance of finding a suitable platform early in the search.

Step B3: Find a contractor that can perform the task. The delegation capability can now iterate over the list of potential contractors in the order determined by the auction.

If the end of the list has already been reached, delegation fails (B3a), causing step A2 in the previous subsection to return failure. This failure is then shown to the operator in step A3.

Otherwise (B3bstep) the next candidate contractor B is chosen from the list and the delegating agent A initiates a delegation

protocol that extends the FIPA Contract Net protocol [57, 58]. On the initiating side, this protocol is handled by a *delegation protocol initiator*. As indicated in the figure, a new initiator is created for every candidate contractor.

The initiator sends a *call-for-proposal (CFP)* message to the delegation capability of the potential contractor (B3c), which in turn causes a *delegation protocol participant* to be created by agent B. The initiator and participant then communicate using additional speech act messages. The participant evaluates the request as described in the next section and replies (to B3d) with a *propose* or *refuse* message.

A *refuse* message (B3e) indicates that the potential contractor cannot perform the task, despite having bid on the task earlier. This can be due to changing circumstances between the auction and the call for proposal, or due to the fact that auctions must be performed quickly and should not necessarily entail a complete verification of the feasibility of a task. Regardless of the reason, delegation will continue by trying to delegate the task to the "next best" agent in the list according to the previous auction (back to step B3a).

A *propose* message (B3f) indicates success: The participant has ensured that each node in the tree being delegated has been provisionally assigned to an agent, which has provisionally reserved the necessary resources for performing the task in a way that satisfies the associated constraints. Given the assumption that we are delegating the root node, the entire TST now has a valid allocation, so the delegation system can signal success to the caller (B3g) by returning a complete proposal. Since we are at the topmost level, this is done by another mechanism than a speech act. The caller could for example be the user interface, in which case step A2 in the previous subsection would now return success and a proposal could be presented to the ground operator in step A4.

The ground operator would eventually respond, causing step A5, A6 or A7 to provide information to step B3h. If the proposal was accepted, the initiator that was created in step B3b sends an *accept-proposal* message to the corresponding participant on the current potential contractor (B3i), and the delegation protocol has finished. Similarly, if the proposal was rejected, the initiator sends a *reject-proposal* message (B3j) and the delegation protocol has finished. Otherwise, we return to step B3c to ask the participant on agent B for an alternative proposal. Note that in this case no new initiator is created. Instead a new *call-for-proposal* message is sent to by the same initiator to the same participant within the same conversation, informing it that an alternative proposal is desired.

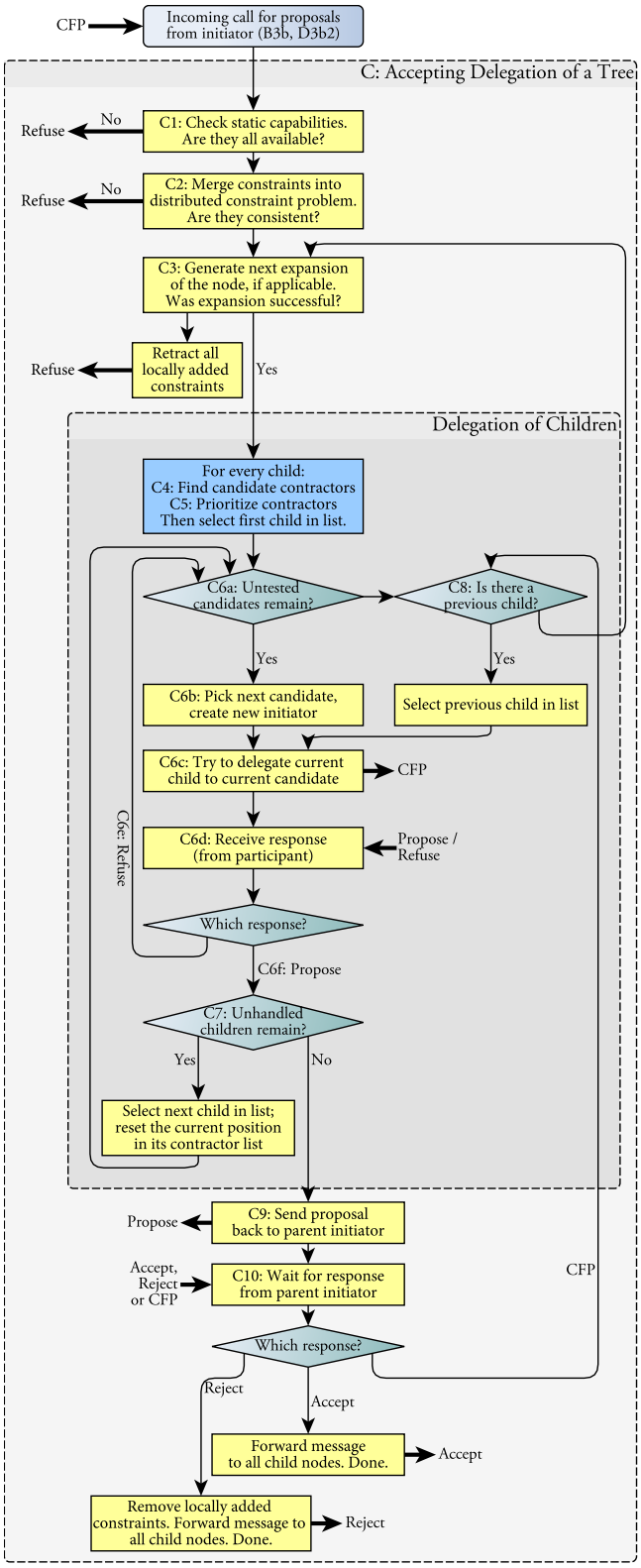

Figure 19. Delegation: Accepting delegation of a node.

6.2.3. *Delegation in the Potential Contractor*

As indicated above (step B3c), a *delegation protocol initiator* for a delegating agent A can send a *call-for-proposal (CFP)* to an agent B asking whether it can be the contractor for a specific delegation $\mathsf{S\text{-}Delegate}(A, B, \tau)$. When the CFP is received, the first step B takes is to create a *delegation protocol participant* that handles all communication with the initiator.

The procedure followed by this participant on the contracting agent is illustrated in Figure 19, where speech act-based communication with the delegator's initiator is shown on the left hand side of each box, while such communication between the contractor and a *sub*-contracting agent is shown on the right hand side. Briefly, the participant must determine whether it believes it can execute the top node n of τ, and if this is the case, reserve the associated resources and verify that all constraints remain satisfiable. Each child of a node must also be recursively delegated to an agent – possibly the same one (B), possibly another. Delegating the children of a node has strong similarities to the delegation process for the root node and will also result in new initiators being created and *call-for-proposal* speech acts being sent to sub-contractors (step C6c below). However, there is a certain degree of added complexity due to the fact that there can be multiple children as opposed to a single root node.

Step C1: Check static capabilities. Though no agent should attempt to delegate a tree or subtree to an agent that lacks the required static capabilities for its root node (ensured in step B1 above), it is nevertheless necessary for the participant to verify this as well, to be certain that it does not agree to a task it cannot perform. If any required capability is missing, the participant immediately refuses the delegation by sending a *refuse* message back to its associated participant (received in step B3d or C6d). This causes the overall delegation process to backtrack, and the caller must attempt to find another agent to which the task can be allocated.

Step C2: Check constraints. The root node n of the tree or subtree being delegated must be integrated with a potentially platform-specific set of constraints related to the type of n, as shown for a **scan-area** action in Section 6.1. If the resulting set of constraints cannot be satisfied for the current task allocation, agent B cannot perform the task ($\neg Can_B(\tau)$), and a *refuse* message is sent. Note that constraints added here are retained in the constraint store for the remainder of the delegation process, until and unless they are actively removed due to backtracking.

Step C3: Expand the node. Since an agent must not accept the delegation of a TST as a contractor unless it has verified that it can actually complete the corresponding simple or complex task, certain additional computations may have to be done during delegation as opposed to execution.

In particular, if n is a *goal node*, B generally cannot verify from the goal formula alone that the goal is achievable or that the actions required to achieve the goal can be performed in a way that satisfies the associated constraints. Therefore it must not accept the delegation until it has called a planner to generated a plan that does achieve the goal, *expanded* the TST by adding these actions as children of the goal node, and successfully delegated all actions to suitable contractors (which may include itself).

Similarly, some mission-specific node types can require other forms of tree expansion before delegation can be accepted, and are considered to be *expandable nodes*. For example, the task of relaying information from a surveillance aircraft to a base station could be expressed as a single node from the point of view of the delegator. A contractor could then choose to view this node as expandable, calling a relay positioning algorithm [59] to generate new flight and relaying actions that would be added to the TST as children to the expandable node.

A node may have multiple expansions – for example, there may exist multiple plans to achieve a particular goal. If delegation returns to step C3 after having tested one or more expansions, the next expansion will be generated.

The process of expanding a node can fail, for example if no (further) solution exists to the planning problem inherent in a goal node. If this happens, the participant retracts the constraints that were added in step C2 and refuses the delegation.

Steps C4, C5: Find and prioritize contractors. These steps are identical to the process followed in steps B1 and B2, but are performed for all child nodes as opposed to a single node. After this has been done, we select the first child in the list of children: The leftmost child of node n in the tree, which is the first child we will attempt to delegate.

Steps C6a–C6f: Try one candidate for the current child. These steps are very similar to the process followed in steps B3a–B3f: The participant iterates over the remaining untested candidate subcontractors for the current child node being processed (C6a). For each candidate subcontractor we create an initiator (C6b), ask for a proposal (C6c) and wait for a response (C6d). A *refuse* message (C6e) indicates that the potential subcontractor cannot perform the task. Delegation will then continue by trying to delegate the task to the "next best" agent in the list. A *propose* message (C6f) instead indicates success, in which case we also have to take care of any remaining children as discussed below.

Step C7: Handle remaining children. If the response from a subcontractor's participant in step C6d was *propose*, we have found a feasible delegation for the current child. Unlike the situation for the root node in step B3f, this does not necessarily mean that we are done: There may be additional child nodes to delegate. If this is the case we step to the next child in the list. We also reset the current position in that child's contractor list, so that we will begin with its first potential contractor. This is important in the case where the child had already been delegated previously, but where we had then backtracked over that delegation as discussed in step C8. We then return to step C6a for another iteration.

Step C8: Backtrack to the previous child. If we have processed the entire list of candidates for a particular child node without finding one to which the node can be delegated, the process does not necessarily fail in step C6a: There may be a possibility to backtrack and find other delegations for the *previous* children of the same node, which may in turn allow this child to be delegated successfully.

If there exists a previous child, we know we have already received a proposal for that child at an earlier stage in the delegation process. We go back one step in the list of children and select it, and then continue to step C6c. This will send a new *call-for-proposal* to the same subcontractor as before, ensuring that this subcontractor has a chance to generate a new proposal if possible. Only if no alternative proposal exists (C6e) do we return to step C6a to test another candidate for the same child.

If there is no previous child, there are two possibilities. First, we may have processed the entire list of potential subcontractors for the first child of the current node without receiving a proposal. Second, we might have found a subcontractor, continued delegating, failed to find subcontractors for later children, and backtracked all the way to the first child. In either case, there are no more delegation alternatives to be explored for the current children of n. The participant therefore returns to step C3 in order to determine whether alternative expansions exist. If n is an expandable node, all generated children are removed and the participant determines whether there is an alternative expansion, in which case the process continues as described earlier. If n is not an expandable node, all locally added constraints are retracted and delegation is refused.

Step C9: Send a proposal. If all children have been delegated, the participant can reply to the initiator with a *propose* message referring to the consistently instantiated TST that has been generated. This is very similar to step B3g, the main differences being that communication takes place using a speech act and that we may only be proposing a solution for a subtree of the complete mission. The recipient of the proposal is either step B3d, if we were delegated the root node, or step C6d, if we were delegated a non-root node.

Step C10: Wait for and handle the response. Recall that the fact that we have proposed a solution does *not* mean that the tree rooted in node n has been completely delegated. The final delegation cannot take place until it has been verified that suitable contractors have been found for the *entire* tree that a ground operator wants to be delegated – and even though the subtree being handled here could be successfully allocated to a certain set of agents (causing us to send back a proposal), this particular assignment of agents and this solution to the constraint problem can in fact cause the *remainder* of the tree to become impossible. For example, assigning a long flight action to a slow aircraft early in the delegation process can make it impossible to find a solution for the entire mission that satisfies all associated temporal deadlines.

Therefore delegation may have to backtrack or backjump even over proposals, asking an agent to generate a new proposal involving a different task allocation or a different solution to the constraint problem. Additionally, even a complete and consistent proposal for an entire mission might not be accepted by the ground operator. Therefore we must now wait for a response to the proposed solution from the initiator, like we did in step B3h. There are three possible responses.

(1) If the response is *accept*, the delegation participant can forward this acceptance to all child nodes (similar to B3i). Then delegation has completed for this tree or subtree and the participant can terminate.
(2) If the response is *reject*, the participant can also forward this message to all child nodes (similar to B3j). Again, delegation has completed and the participant can terminate.
(3) We may also find out that we have to backtrack the delegation process for this node despite having generated a complete proposal. This can be because other parts of the process did not find contractors for *later* nodes in the tree given the current expansion and task allocation for this subtree, or due to the ground operator rejecting the current proposal. In this case the response from the parent initiator will be another *call-for-proposal* message within the same conversation, which is interpreted as a request for an alternative proposal.
The participant returns to step C8. If the node does have children, it steps back from its current position *after* the end of the child list to the last child, and attempts to find a new delegation for that child. This ensures that all alternatives for child nodes are explored correctly. If the node does not have children, or if all alternatives for the children have already been explored, step C8 returns to step C3 where we may instead attempt to generate an alternative expansion.

Delegation, backtracking and backjumping. As the delegation process is described above, chronological backtracking is used. We have also extended this process to use *backjumping*, which can skip over multiple nodes in a single step [60]. This process is exemplified informally below.

6.3. *Delegation Process Example*

We will now consider the delegation process in the context of the supply delivery scenario discussed in Section 5. The assumption is that several survivors have been located after a natural disaster, and that a mission of delivering emergency supplies to these survivors has been specified, resulting in the unallocated and undelegated TST shown in Figure 20. This TST contains a sub-TST (N_1–N_{12}) for loading a carrier with four boxes (N_2–N_6), flying the carrier to a location close to several survivors (N_7), and unloading the packages from the carrier in order to deliver them (N_8–N_{12}). A single box must also be delivered to a lone survivor in another region, far away from where most of the survivors were found (N_{13}). These two aspects of the mission are comparatively independent and can take place concurrently given a sufficient number of platforms. Several packages on a carrier can also be both loaded and unloaded concurrently, but the loading, moving, and unloading of the carrier is a sequential operation.

We assume that we have a single carrier at our disposal and that there are six agents in the vicinity: Four UAV platforms/agents (P_1–P_4) and two agentified ground stations (GS_1 and GS_2). Another ground operator is already using GS_2 to perform a mission with the platforms P_3 and P_4 in an area nearby. We are therefore using the ground station GS_1 and will mainly use the platforms P_1 and P_2 to achieve our mission. We may also be able to borrow P_3, which is currently idle, but this is associated with an additional cost.

Using the graphical interface on GS_1, we indicate that we want the TST to be delegated. The ground station is thus the agent assigned as responsible for delegating the TST, and it asks its own delegation capability to initiate delegation for the TST (step A2 above).

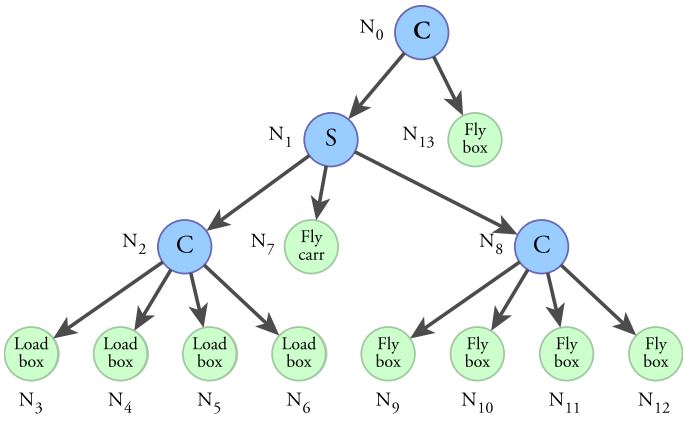

Figure 20. A TST for supply delivery.

The delegation capability on GS_1 searches for a platform to which it can delegate the TST (step B1). It starts by finding all agents that have the required static capabilities for the top node N_0, which would include all agents. It then auctions out N_0 to find the best initial choice (step B3). In this case, the specified marginal cost is the lowest for P_1 and P_2 (which are free), somewhat higher for the ground stations (which are normally not involved in executing such nodes) and even higher for P_3 and P_4 as they are assigned to another ground operator. The first platform, P_1, is chosen as the winner (steps B3a and B3b) and is sent a *call-for-proposal* message for N_0 (step B3c).

The delegation capability of P_1 receives a request to be the contractor for N_0 (Figure 19). It verifies that it can execute concurrency nodes (C1), adds the appropriate constraints to its constraint store (C2), and verifies that they are satisfiable. Concurrency nodes cannot be expanded (C3). It must then attempt to recursively delegate all child nodes.

Candidate contractors are found and prioritized (steps C4 and C5). Delegation of child nodes then occurs in left-to-right order. P_1 therefore asks its own delegation capability to initiate delegation for the first child node, N_1. In this case, P_1 is again chosen as the best contractor as it is one of the agents having the lowest marginal cost and we prioritize executing child nodes on the same agent. In step C6c, P_1 therefore sends a *call-for-proposal* for N_1 to itself. A new instance of the delegation protocol results, where node N_1 is the target – essentially, one instance of Figure 19 is now communicating with another instance of the same figure. The new participant verifies that it has the static capabilities for N_1 (C1), extends the constraint network (C2) and verifies that it remains satisfiable. This must be the case, because the nodes added so far only contain temporal constraints and because there are no elementary actions requiring time for their execution. After verifying that this node is not expandable (C3), P_1 once more has to delegate all child nodes.

The treatment of N_1's first child N_2 is identical, resulting in another level of recursion. Thereafter, the first child N_3 is an elementary action node, where the choice of platform is the key to a successful allocation due to each platform's unique state, constraint model for the action, and available resources. The candidates for node N_3 are platforms P_1–P_4. At this point P_1 happens to be the closest to the package depot and therefore gives the best bid for the node. A *call-for-proposal* is again sent from P_1 to itself. Upon receiving the call-for-proposal, it verifies that it can execute the node and replies with a *propose* message.

We have now returned to the delegation of N_2, where the second child, N_4, has to be delegated. For this node, 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 for N_5 and N_6. The schedule initially defined by nodes N_0–N_2 is now also constrained by how long it takes for P_1 and P_2 to carry out action nodes N_3–N_6, and the constraints involved are distributed among platforms P_1 and P_2.

We have now returned to the delegation of N_1, where the next child for P_1 to delegate is the carrier delivery node N_7. The only platform that has the capabilities for this task is P_1, and it is delegated the node. Continuing with 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 delegated all the nodes N_8–N_{12}.

The final node, N_{13}, is delegated to platform P_2 and the delegation is complete. The total mission time is 58 minutes, which is much longer than the operator expected. Since the constraint problem defined by the allocation of the TST is distributed between the platforms, it is possible for the operator to modify the constraint problem by adding more constraints, and in this way modify the resulting task allocation. The operator puts a time constraint on the mission, restricting the total time to 30 minutes.

To re-allocate the TST with the added constraint, GS_1 sends a *reject-proposal* to platform P_1. The added time constraint makes the current allocation inconsistent, so simply finding a new constraint solution is not possible. The last allocated node must therefore be re-allocated. However, no platform for N_{13} can make the allocation consistent, not even the unused platform P_3. Backtracking starts [7]. 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 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. 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 deduce 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. However, removing the allocations for the 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 makes 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 does not have enough time to deliver the final package. Instead, a new backjump point search starts, finding node N_5. The search continues from N_5. This time, 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}. This allocation is consistent. The delegation algorithm finishes on platform P_1, by sending another *propose* message back to the operator. The operator can then approve the allocation and starts the mission. An *accept-proposal* message is sent to P_1, and similar messages are recursively sent down throughout the tree, after which execution can start.

7. Delegation, Planning, and Distributed Planning

When discussing an example logistics mission in Section 5.4, we described one way in which a ground operator can receive assistance from an automated planner: The mission-specific ground operator interface generates a planning problem corresponding to the mission at hand, makes a direct call to the planning capability on board the ground station, receives a complete plan in the shape of a mission specification, converts the entire plan to a TST, and then delegates this TST to one or more contractors that will carry it out. We will now demonstrate in three distinct steps how we can benefit from a closer integration between planning and delegation.

7.1. *Planning for Goal Nodes*

The most straight-forward way to achieve integration is through goal nodes. Instead of calling the planner itself, the mission-specific ground operator interface calls the TST Factory to generate a TST consisting of a single goal node, which refers to the problem instance specification. It then asks the delegation capability to initiate delegation for the root node of this small TST (Section 6.2.2). As part of step B1, delegation broadcasts a query for agents having a general planning capability, which is a requirement for taking on the responsibility for a goal node. Since the ground station on which the ground operator interface is running is in itself an agent, it may reply to its own request. This allows all types of delegation to be handled coherently and consistently, without special cases.

After an auction (step B2), one of the responding agents will be sent a *call-for-proposal* message (step B3c). This potential contractor must determine whether it can perform the task inherent in the goal node. This does not merely mean that the agent has a planner and can make an attempt to generate a plan (static capabilities, step C1) – it means that the agent can *succeed* in finding a plan and can ensure that the plan is *executed*. Therefore the delegation request cannot immediately be accepted. Instead the agent must call its own on-board task planner and determine what an appropriate plan for the given goal node would be (part of the node expansion in step C3).

Once a plan is generated, the node expansion procedure converts it to a TST and attaches this plan TST as a child beneath the goal node. Delegation automatically continues to ensure that the newly generated plan nodes are recursively delegated (steps C4–C8). If this fails, one can consider generating an alternative plan (back to step C3). If the planner finds no alternative plans, or is configured to only generate a single plan, delegation will be *refuse*:d. Only if a plan has been both generated and successfully provisionally delegated does the contractor reply with a *propose* message (step C9).

When considered in the context of a trivial tree consisting of a single goal node, the benefit of this procedure may not be immediately obvious. There is however a significant advantage in the ability to place one or more goal nodes at any point in a TST, allowing a flexible mix between mission aspects that are specified in detail and those where a great deal of freedom is given to contractors.

7.2. *Interleaving Planning and Delegation*

In the procedure described above, planning and delegation are interleaved in a comparatively coarse-grained manner: Every time a goal node is encountered, a complete plan is generated before delegation resumes. If resources are tight and the first solution plan is not feasible, a new complete alternative plan is generated, after which delegation is restarted from the beginning of the new plan. We can benefit significantly from instead integrating these two processes at a fine-grained level where the delegation process actively "drives" the planning process forward one step at a time, choosing when and where an attempt should be made to provisionally delegate the current partially generated plan.

First, if such a partial delegation attempt *succeeds*, all resources involved will be provisionally booked for the remainder of the plan generation phase, avoiding a race condition where a plan could be feasible during its generation but become infeasible before it is delegated.

Second, if the partial delegation attempt *fails*, the planner can be informed. This results in early and frequent feedback to the planning algorithm, allowing it to immediately detect *where* its search results in an infeasible partial plan and to backtrack and find alternative solutions. Problems are then detected for each action instead of only for an entire solution plan, which can significantly improve performance. This use of delegation can be seen as providing indirect access to information that is difficult or impossible to integrate in a planning problem description, information that includes but is not limited to the following:

(1) *When* can a particular agent schedule a new action? Note that the answer is not necessarily a set of fixed intervals in time: The schedule of an agent could correspond to a flexible temporal network where already contracted tasks can be adjusted somewhat in time in order to allow new tasks to be accepted.
(2) Where will the agents be at specific points in time according to the tasks they have already contracted? This is important in order to determine how much time is required for a **fly-to** action, for example.
(3) Exactly how much time will an agent require in order to execute an action? The time requirements for a **fly-to** action are not only dependent on the distance but also on how the platform in question generates flight paths using motion

planning and on the flight envelope of the platform, which a planner on another platform does not necessarily have access to.

(4) Which other platform-specific and node-specific constraints must be satisfied for delegation to succeed?

To simplify the presentation we will assume the planner is initially provided with identities for all potential contractors, so that one action thread can be generated for each contractor. As there is no requirement for all of these contractors to be available throughout the planning process, this is a very minor restriction. Interleaving planning with delegation then requires a modified planning and delegation procedure that essentially integrates steps C3 (generate the next expansion) and C4–C8 (delegate child nodes) in Figure 19. This procedure can be described as follows.

procedure TFPOP-delegation
  $\pi \leftarrow \langle \{a_0\}, \varnothing, \varnothing, \varnothing \rangle$ *// Initial plan*
→**generate** skeleton TST
→**delegate** skeleton TST
→**if** delegation of skeleton TST fails **then fail**
  repeat
→ **if** goal satisfied **then**
→ **propose** (step C9)
→ **wait** for response (step C10)
→ **if** response is *accept-proposal* **then success**
  choose a thread t to which an action should be added
  // Use partial state to filter out most potential actions
  $s \leftarrow$ partial state at the end of thread t
  choose an action a for t such that $eval(pre(a)) \neq false$
  if effects of a are inconsistent **then fail** (backtrack)
  // Complete check:
  // Can the action really be added, and how?
  choose precedence constraints C and causal links L
    using *make-true*(), ensuring $pre(a)$ is satisfied,
    a does not interfere with existing actions
    and no existing action interferes with a
  update resource usage
  if resource constraints violated **then fail** (backtrack)
  add a, C, L and necessary mutex sets to π
  update existing partial states
  create new partial state for a
→ add TST node corresponding to a
→ **foreach** $c \in C$: Add corresponding constraint in TST
→ **delegate** newly generated nodes
→ **if** delegation fails **then fail** (backtrack)

As in Section 5.2, **choose** denotes standard non-deterministic choice and is implemented through backtracking. Every time the procedure backtracks over the addition of an action a and a set of constraints C, the existing delegation of the corresponding TST node is rejected, causing the associated participant to retract the corresponding constraints from the TST.

The following modifications can be noted compared to the original TFPOP planner, marked with arrows above.

First, before the search process begins, the planner must use the TST factory to generate an initial skeleton TST. In sequential planning this would consist of a single sequence node under which all actions can be added. For TFPOP, there is a single concurrency node under which there is one sequence node for each thread (potential contractor). This initial TST must be provisionally delegated to appropriate agents. If delegation fails, which is unlikely given that the tree only consists of control nodes, the entire planning process fails. This corresponds to failure in step C3 (Figure 19) and causes the participant for the goal node to refuse the delegation.

Second, if the planner determines that its current plan candidate satisfies the specified goals, then all actions involved in achieving those goals (as well as all required control structures such as sequence nodes) have already been provisionally delegated and distributed to one or more executing agents. Therefore no plan has to be returned. Instead, this corresponds directly to the case where no more children remain in step C7. The planner asks the goal node's participant to send a proposal to the parent initiator (step C9). It then waits for a response (C10). If the proposal was accepted, the planner has succeeded and can terminate, allowing the participant to forward the *accept-proposal* message to the nodes generated by the planner. Otherwise, the planner can continue to generate an alternative proposal.

Third, each time an action is added to the plan, the planner must call the TST factory to generate a corresponding elementary action node and attach it under the agent's sequence node. The associated new precedence constraints must also be added to the TST. The new elementary action is then provisionally delegated. Since an executing agent has already been chosen by TFPOP, there is no need to iterate over potential contractors. Furthermore, there is only one new child to delegate. Delegation therefore only requires creating one new initiator, as in step C6b in Figure 19. The initiator then sends a *call-for-proposal* message to the intended agent (C6c), which creates a participant that verifies the executability of the action and the satisfiability of the extended constraint problem. Eventually a response arrives (C6d).

If delegation succeeded, the contractor has added the appropriate constraints to its constraint store and ensured that the required capabilities and resources are available and provisionally reserved. Otherwise the planner can immediately backtrack, trying another action to extend the plan.

Given this considerably tighter integration between planning and delegation, the action preconditions that are included in a planning domain description can be limited to those that are the most useful in pruning the search tree, which greatly simplifies the work of the knowledge engineer. The delegation functionality can then verify the stronger and potentially platform-specific conditions associated with each TST node, resulting in a division of responsibilities where planning and delegation processes work in concert to efficiently determine executability.

7.3. *Distributed Planning through Delegation*

The integrated process described above can be viewed as leading to a specific form of distributed planning, where different goal nodes specified within a single TST can be expanded into plans by different agents on board different platforms. This form of distribution is applicable when an overall mission is naturally

separable into several specific planning problems. It also forms an extensible foundation on which we can now easily construct a truly distributed planner, where several agents can cooperate in elaborating a single goal node into a task specification tree grounded in the elementary actions available on each platform involved in its execution.

To this effect, we extend the integration one step further (Figure 21). Assume a ground operator has specified a mission using a TST containing one or more goal nodes (D1). For example, let us assume there is only a single goal node specifying the goal for an emergency services logistics mission where crates of supplies should be transported to injured people. The goal TST arrives at the delegation subsystem (D2) of agent *A*, which eventually extracts a goal formula from the goal node and sends it to the integrated TFPOP task planner (D3).

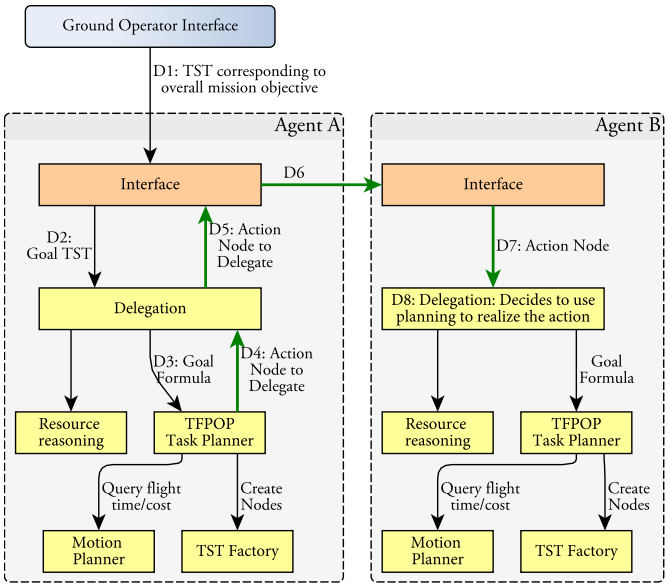

Figure 21. Delegation and planning.

TFPOP begins as described in the previous subsection, generating one action at a time and delegating it to an appropriate agent. One of the generated actions loads an entire carrier with a set of suitable supply crates. To the planner of agent *A*, this is an ordinary single-agent action with ordinary effects. The planner therefore generates a TST node for the action and asks the delegation subsystem to delegate it (D4). The delegation subsystem communicates with agent *B* through its interface (D5), sending a *call-for-proposal* speech act (D6) for the newly generated action node. The interface on agent *B* receives this speech act and forwards it to its own delegation subsystem (D7, D8).

Here, agent *B* decides that it would like to use its own onboard planner to elaborate the task into what it considers to be elementary actions. It therefore views the node not as an elementary action but as a goal to be achieved, generates a suitable planning problem specification, and calls TFPOP. Agent *B* can then use its more detailed knowledge about its own capabilities and its own schedule to elaborate the task into a concrete tree of elementary action nodes.

This procedure can of course be applied in more than one level: To achieve its "second-level" goal, agent *B* may generate an action that is delegated to agent *C*, which views it as a third-level goal to be achieved. The result is a distributed planner based on delegation, which has certain hierarchical aspects but does not require the specification of a hierarchy in advance. Indeed, there is no predefined direction in which actions may be delegated, and agent *B* or *C* may end up delegating actions back to agent *A*.

8. Related Work

Due to the multi-disciplinary nature of the work considered here, there is a vast amount of related work too numerous to mention in complete detail. In addition to the work referenced in the article, we instead consider a number of representative references from the relevant research areas.

In terms of formal mission specification languages, logics of action and change have been studied extensively [61–66], yet there has been relatively less research which focuses specifically on composite actions in all their generality. An early precursor is the use of strategies in the situation calculus [67]. A more extensive and recent study of composite actions is the Golog [68] framework which is also based on a dialect of the situation calculus [64,69]. Davis and Morgenstern [70] also propose a formal logical theory that has some relation to ours in the sense that they use speech acts for communication in the context of multi-agent planning.

Specifying delegation as a speech act as we do in this article is novel. Castelfranchi and colleagues have previously studied the concept of delegation from the social science perspective [71]. In terms of formal specification, besides the work here, Lorini et al [72] propose a logical formalism that models the intentions and beliefs of a delegating agent based on Castelfranchi's work.

The concept of autonomy has a long and active history in multi-agent systems [73, 74]. One early driving force was space missions that focused on the problem of interaction with autonomous agents and the adjustability of this autonomy [75, 76]. Later, Hexmoor and McLaughlan argue that reasoning about autonomy is an integral component of collaboration among computational units [77]. Hexmoor also argues that trust is essential for autonomy [78]. According to his definition, the autonomy of an agent *A* with respect to a task *t* is the degree of self-determination the agent possesses to perform the task. This is similar to the view on autonomy in our approach, where the level of autonomy for an agent is dependent on the strictness of the constraints on the tasks that are delegated to the agent.

Cooperative multi-robot systems have a long history in robotics, multi-agent systems and AI in general. One early study presented a generic scheme based on a distributed plan merging process [79], where robots share plans and coordinates their own plans to produce coordinated plans. In our approach, coordination is achieved by finding solutions to a distributed constraint problem representing the complex task. Another early work is ALLIANCE [80], which is a behavior-based framework for instantaneous task assignment of loosely coupled subtasks with or-

dering dependencies. Each agent decides on its own what tasks to do based on its observations of the world and the other agents. Compared to our approach, this is a more reactive approach which does not consider what will happen in the future.

M+ [81] integrates mission planning, task refinement and cooperative task allocation. It uses a task allocation protocol based on the Contract Net protocol with explicit, pre-defined capabilities and task costs. A major difference to our approach is that in M+ there is no temporally extended allocation. Instead, robots make incremental choices of tasks to perform from the set of executable tasks, which are tasks whose prerequisite tasks are achieved or underway. The M+CTA framework [82] is an extension of M+, where a mission is decomposed into a partially ordered set of high-level tasks. Each task is defined as a set of goals to be achieved. The plan is distributed to each robot and task allocation is done incrementally as in M+. When a robot is allocated a task, it creates an individual plan for achieving the task's goals independently of the other agents. After the planning step, robots negotiate with each other to adapt their plans in the multi-robot context. Like most negotiation-based approaches, M+CTA first allocates the tasks and then negotiates to handle coordination. This is different from our approach which finds a valid allocation of all the tasks before committing to the allocation.

ASyMTRe [83], uses a reconfigurable schema abstraction for collaborative task execution providing sensor sharing among robots, where connections among the schemas are dynamically formed at runtime. The properties of inputs and outputs of each schema is defined and by determining a valid information flow through a combination of schemas within, and across, robot team members a coalition for solving a particular task can be formed. Like ALLIANCE, this is basically a reactive approach which considers the current task, rather than a set of related tasks as in our approach. IQ-ASyMTRe [84] is a recent extension of ASyMTRe that handles tightly coupled multirobot tasks involving close robot coordinations. Other Contract-Net and auction-based systems similar to those described above are COMETS [51], MURDOCH system [49], Hoplites [85] and TAEMS [86].

Many task allocation algorithms are, as mentioned above, auction-based [49, 52–54, 85, 87, 88]. 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 unrelated utilities guarantee that each task can be treated as an independent entity, and can be auctioned out without affecting other parts of the allocation. This means that a robot does not have to take other tasks into consideration when making a bid.

More advanced auction protocols have been developed to handle dependencies between tasks. These are constructed to deal with complementarities. Examples are sequential single item auctions [89] and combinatorial auctions [90]. These auctions typically handle the situation where 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 [89] is of special interest since it is similar to our approach. In SSI auctions, as in our task allocation approach, tasks are auctioned out in sequence, one at a time to make sure the new task fits with the previous allocations. The difference is what happens when there is no agent that can accept the next task. In SSI auctions common strategies are to return a task in exchange for a new task or to start exchanging tasks with other agents. This is basically a greedy approach which is incomplete. Our approach on the other hand uses backtracking which is a complete search procedure. Normally SSI auctions are applied to problems where it is easy to find a solution but it is hard to find a good solution. When allocating the tasks in a TST it is often hard to find any solution and SSI auctions are therefore not appropriate.

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 [90]. Since all agents have to bid on all bundles (in our case tasks) they could accept in one round, it means that even in the best case there is a very high computational cost involved in using combinatorial auctions. Another weakness is that they do not easily lend themselves to a recursive process where tasks are recursively decomposed and allocated. Our approach, on the other hand, is suitable for recursive allocation and by using heuristic search tries the most likely allocations first which should result in much better average case performance.

Automated planning is also an essential aspect of missions as described in this article. Whereas a single-agent planner has to decide what actions to perform to achieve a given goal, a multi-agent planning system must also determine how to distribute actions and subtasks across a set of homogeneous or heterogeneous agents in order to make the best possible use of all available resources.

Many approaches rely on the simplifying assumption that a specialized coordinator has detailed knowledge about which agents are available, what actions they can perform and what resources are available, as well as complete authority to assign detailed plans to arbitrary agents. All agents are also assumed to share a common execution architecture supporting a particular plan structure. Under these very strong assumptions, one can create a global planning model incorporating all available agents, reducing the problem of distributing subtasks and coordinating resource usage into a standard planning problem. The planning process is then completely centralized and performed by the coordinator, after which suitable subplans are extracted and distributed to each agent.

Depending on the desired plan structure, a wide variety of planning techniques may be used, including classical planners with support for concurrency [91–95], planners using rich domain information for guidance [37, 96–98], specialized techniques for centralized multi-agent planning [99, 100], and planners based on policy generation using MDPs, Multi-agent MDPs, POMDPs, and decentralized (PO)MDPs [101–107]. As a variation on this theme, the central coordinator may distribute high-level abstract actions to perform, which can then be elaborated into detailed plans by individual agents [108]. Common to these approaches is the lack of any explicit notion of delegation,

negotiation or adjustable autonomy. Additionally, the fact that all information about the feasibility of a given action for a given action must be available centrally can be problematic in many situations.

In an alternative approach, planning can be completely distributed among a set of cooperating agents, all of which are considered to be equals. Coordination among agents may be interleaved with planning [109–112], or may take place after each agent has constructed a complete plan for its own goal [113–115]. Research in this area often focuses on techniques for successive refinement of the local plans generated by each individual agent as it incrementally receives information about plans generated by other agents. The final result is a set of individual plans, each taking care to avoid interference with other agents. Again, distributed planners tend to lack explicit support for delegation or adjustable autonomy, though certain forms of negotiation are occasionally used.

One of the earliest detailed accounts of issues involving mixed-inititive planning is [116]. More recently, a number of attempts have been made to develop mixed-initiative planning systems. Two representative examples are Rationale [117] and MapGen [118]. The latter is particularly interesting since it was developed by NASA and used in the Mars Rover Project. Interfaces used with mixed-initiative planning systems are another active area of research. Horvitz [119] provides an overview of principles and Finzi [120] proposes mixed-initiative techniques for human robot interaction in rescue scenarios. The framework described in this article is unique in that it attempts to combine a mixed-initiative mission planning system with a delegation framework.

9. Conclusions

Developing robust, pragmatically useful software systems that can be integrated with existing UASs and that support the specification of high-level collaborative missions, the generation of distributed plans associated with such missions and their execution on teams of UASs, is a highly complex endeavour. Due to this complexity, using formal specifications to guide the development of both the architectural components and the functionalities that are included in the architecture is a necessity.

In this article, we have shown how this can be done. Each of the fundamental components in the framework we propose begins with formal specifications in temporal and modal logics. In the case of mission specifications, we use TAL, a highly expressive temporal logic of action and change. In the case of TFPOP, the automated planner, the semantic characterization of a plan uses TAL and the output of the planner is a TAL narrative. In the case of the specification of task specification trees, we show how they are formally related to composite actions in TAL and how one can map back and forth between them. In the case of the delegation concept itself, we use a multi-modal logic KARO to specify a new speech act. Tasks themselves are extended to be contextual in nature by associating a set of constraints with each. These constraints can be characterized using TAL or using any other logical formalism.

Since TSTs and constraints are used as a basis for execution of plans in the actual UAS system architectures, in some sense, the formal verifiability of not only the mission plans but of their execution is built into the actual processes associated with the delegation architecture. This is done purposely. New tools and techniques have to be found to be able to provide verification and validation tools for the complex types of functionality required to integrate high autonomy features into robotic architectures such as the components investigated in this paper. We believe the approach described in this article is a viable means of doing this.

In this manner, not only can we build complex systems such as that described here, but we can control the complexity of the implementation process through the use of these formal techniques. This mix of pragmatics with formal specification has resulted in the collaborative UAS system architecture described in this paper. The system is deployed in prototype and is operational on several of our UAS platforms.

The framework, system and functionalities described here are generic in nature. They can be used on different types of robotic platforms due to the separation of the architectural extensions described here and legacy systems associated with existing robotic systems. Additionally, the mission specification and planning techniques can be instantiated to different application areas through the specific choice of plan operators and primitive actions one might want to associate with the individual heterogeneous robotic systems participating in the application.

Acknowledgments

This work is partially supported by the Swedish Research Council (VR) Linnaeus Center for Control, Autonomy, and Decision-making in Complex Systems (CADICS), the ELLIIT network organization for Information and Communication Technology, the Swedish National Aviation Engineering Research Program NFFP5, SSF – the Swedish Foundation for Strategic Research (CUAS Project), the EU FP7 project SHERPA, grant agreement 600958, and CENIIT, the Center for Industrial Information Technology.

Bibliography

[1] P. Doherty, P. Haslum, F. Heintz, T. Merz, T. Persson and B. Wingman, A distributed architecture for intelligent unmanned aerial vehicle experimentation, *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, (2004).

[2] M. Wzorek, G. Conte, P. Rudol, T. Merz, S. Duranti and P. Doherty, From motion planning to control – a navigation framework for an unmanned aerial vehicle, *Proceedings of the 21st Bristol International Conference on UAV Systems*, (2006).

[3] P. Rudol and P. Doherty, Human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery, *Proceedings of the IEEE Aerospace Conference*, (2008), pp. 1–8.

[4] G. Conte and P. Doherty, Vision-based unmanned aerial vehicle navigation using geo-referenced information,

EURASIP Journal of Advances in Signal Processing **2009**(1) (2009).

[5] P. Rudol, M. Wzorek and P. Doherty, Vision-based pose estimation for autonomous indoor navigation of micro-scale unmanned aircraft systems, *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (May 2010), pp. 1913–1920.

[6] P. Rudol, Increasing autonomy of unmanned aircraft systems through the use of imaging sensors, licentiate thesis, Department of Computer and Information Science, Linköpings universitet (2011). Linköping studies in science and technology. Thesis: 1510.

[7] P. Doherty, F. Heintz and D. Landén, A delegation-based architecture for collaborative robotics, *Agent-Oriented Software Engineering XI: Revised Selected Papers*, eds. D. Weyns and M.-P. Gleizes, *Lecture Notes in Computer Science* **6788** (Springer-Verlag, Berlin Heidelberg, 2011), pp. 205–247.

[8] P. Doherty and J.-J. C. Meyer, On the logic of delegation: Relating theory and practice, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, eds. F. Paglieri, L. Tummolini and R. Falcone (College Publications, London, 2012).

[9] C. Castelfranchi and R. Falcone, Toward a theory of delegation for agent-based systems, *Robotics and Autonomous Systems*, **24** (1998), pp. 141–157.

[10] R. Falcone and C. Castelfranchi, The human in the loop of a delegated agent: The theory of adjustable social autonomy, *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans* **31**(5) (2001) 406–418.

[11] P. Cohen and H. Levesque, Intention is choice with commitment, *Artificial Intelligence* **42**(3) (1990) 213–261.

[12] J. L. Austin, *How to do things with words* (Harvard University Press, 1975).

[13] J. R. Searle, *Speech acts: An essay in the philosophy of language* (Cambridge university press, 1969).

[14] P. Doherty and J.-J. C. Meyer, Towards a delegation framework for aerial robotic mission scenarios, *Proceedings of the 11th International Workshop on Cooperative Information Agents (CIA)*, (2007).

[15] W. van der Hoek, B. van Linder and J.-J. C. Meyer, An integrated modal approach to rational agents, *Foundations of Foundations of Rational Agency*, eds. M. Wooldridge and A. Rao, *Applied Logic Series* **14** 1998.

[16] M. Wzorek and P. Doherty, Reconfigurable path planning for an autonomous unmanned aerial vehicle, *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, (2006), pp. 438–441.

[17] M. Wzorek, J. Kvarnström and P. Doherty, Choosing path replanning strategies for unmanned aircraft systems, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, (2010).

[18] M. Ghallab, On chronicles: Representation, on-line recognition and learning, *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, (1996), pp. 597–607.

[19] F. Heintz and P. Doherty, Chronicle Recognition in the WITAS UAV Project: A Preliminary Report, *Proceedings of the Swedish AI Society Workshop (SAIS)*, (2001).

[20] F. Heintz, J. Kvarnström and P. Doherty, A Stream-Based Hierarchical Anchoring Framework, *Proceedings of the IEEE/RSJ International Conference on Intelligent RObots and Systems (IROS)*, (IEEE conference proceedings, 2009), pp. 5254–.

[21] F. Heintz, J. Kvarnström and P. Doherty, Stream-Based Hierarchical Anchoring, *Künstliche Intelligenz* (2013).

[22] FIPA-ACL, *FIPA Communicative Act Library Specification*. Foundation for Intelligent Physical Agents, (2002).

[23] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, Ros: An open-source robot operating system, *ICRA workshop on open source software*, (2009).

[24] M. Wzorek, Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, licentiate thesis, Linköpings universitet (2011).

[25] J. Kvarnström and P. Doherty, Automated planning for collaborative systems, *Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV)*, (2010).

[26] J. Kvarnström, Planning for loosely coupled agents using partial order forward-chaining, *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, eds. F. Bacchus, C. Domshlak, S. Edelkamp and M. Helmert (2011).

[27] P. Doherty, J. Gustafsson, L. Karlsson and J. Kvarnström, (TAL) temporal action logics: Language specification and tutorial, *Electronic Transactions on Artifical Intelligence* **2**(3-4) (1998) 273–306.

[28] P. Doherty and J. Kvarnström, Temporal action logics, *The Handbook of Knowledge Representation*, eds. V. Lifschitz, F. van Harmelen and F. Porter (Elsevier, 2008), pp. 709–757.

[29] P. Doherty and J. Kvarnström, Tackling the qualification problem using fluent dependency constraints, *Computational Intelligence* **16**(2) (2000) 169–209.

[30] J. Gustafsson and J. Kvarnström, Elaboration tolerance through object-orientation, *Artificial Intelligence* **153**(1-2) (2004) 239–285.

[31] P. Doherty, J. Kvarnström and F. Heintz, A Temporal Logic-based Planning and Execution Monitoring Framework for Unmanned Aircraft Systems, *Autonomous Agents and Multi-Agent Systems* **19**(3) (2009) 332–377.

[32] P. Doherty, J. Kvarnström and A. Szalas, Temporal Composite Actions with Constraints, *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, (2012), pp. 478–488.

[33] A. Arnold and D. Niwiński, *Rudiments of μ-calculus*, Studies in logic and the foundations of mathematics,, Vol. 146 (Elsevier, 2001).

[34] P. Doherty, D. Landén and F. Heintz, A distributed task specification language for mixed-initiative delegation, *Proceedings of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, (2010).

[35] D. Landén, F. Heintz and P. Doherty, Complex task allocation in mixed-initiative delegation: A UAV case study (early innovation), *Proceedings of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, (2010).

[36] R. W. Weyhrauch, Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* **13**(1–2) (1980) 133–170.

[37] J. Kvarnström and P. Doherty, TALplanner: A temporal logic based forward chaining planner, *Annals of Mathematics and Artificial Intelligence* **30** (June 2000) 119–169.

[38] P. Doherty and J. Kvarnström, TALplanner: A temporal logic based planner, *Artificial Intelligence Magazine* **3** (2001).

[39] J. Kvarnström, Planning for loosely coupled agents using partial order forward-chaining, *Proc. SAIS*, (May 2010).

[40] F. Bacchus and F. Kabanza, Using temporal logics to express search control knowledge for planning, *Artificial Intelligence* **116**(1-2) (2000) 123–191.

[41] D. S. Weld, An introduction to least commitment planning, *AI magazine* **15**(4) (1994) p. 27.

[42] P. Morris, N. Muscettola and T. Vidal, Dynamic control of plans with temporal uncertainty, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, (2001), pp. 494–502.

[43] F. Bacchus and M. Ady, Precondition control, `http://www.cs.toronto.edu/~fbacchus/Papers/BApre.pdf` (1999).

[44] M. Yokoo and K. Hirayama, Distributed constraint satisfaction algorithm for complex local problems, *Proceedings of the International Conference on Multi Agent Systems (ICMAS)*, (1998), pp. 372–379.

[45] K. Marriott, N. Nethercote, R. Rafeh, P. Stuckey, M. G. de la Banda, and M. Wallace, The design of the Zinc modelling language, *Constraints* **13**(3) (2008) 229–267.

[46] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck and G. Tack, Minizinc: Towards a standard CP modelling language, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, (2007).

[47] I. P. Gent, C. Jefferson and I. Miguel, Minion: A fast scalable constraint solver, *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, (IOS Press, 2006), pp. 98–102.

[48] M. Yokoo, Asynchronous Weak-commitment Search for Solving Distributed Constraint Satisfaction Problems, *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, (Springer-Verlag, London, UK, 1995), pp. 88–102.

[49] B. Gerkey and M. Mataric, Sold!: Auction methods for multi-robot coordination, *IEEE Transactions on Robotics and Automation* **18**(5) (2002) 758–768.

[50] B. Gerkey, On multi-robot task allocation, PhD thesis, University of Southern California (2003).

[51] T. Lemaire, R. Alami and S. Lacroix, A distributed tasks allocation scheme in multi-UAV context, *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (2004), pp. 3622–3627.

[52] R. Zlot and A. Stentz, Complex task allocation for multiple robots, *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (2005), pp. 1515–1522.

[53] R. Zlot, An auction-based approach to complex task allocation for multirobot teams, PhD thesis, Carnegie Mellon University (2006).

[54] 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** (2010) 1–23.

[55] S. Parsons, J. A. Rodríguez-Aguilar and M. Klein, Auctions and bidding: A guide for computer scientists, *ACM Computing Surveys* **43**(2) (2011) p. 10.

[56] M. J. Atallah and M. Blanton, *Algorithms and Theory of Computation Handbook* (Chapman & Hall/CRC, 2009), ch. Auction Protocols, 2nd edn.

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

[58] Foundation for Intelligent Physical Agents, FIPA Contract Net Interaction Protocol Specification `http://www.fipa.org`, (2002).

[59] O. Burdakov, P. Doherty, K. Holmberg, J. Kvarnström and P.-M. Olsson, Relay Positioning for Unmanned Aerial Vehicle Surveillance, *The International Journal of Robotics Research* **29**(8) (2010) 1069–1087.

[60] D. Landén, Complex Task Allocation for Delegation: From Theory to Practice, licentiate thesis, Linköpings universitet (2011).

[61] M. Gelfond and V. Lifschitz, Representing action and change by logic programs, *Journal of Logic Programming* **17** (1993) 301–321.

[62] E. Sandewall, *Features and Fluents. The Representation of Knowledge about Dynamical Systems* (Oxford University Press, 1994).

[63] M. Shanahan, *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* (MIT Press, 1997).

[64] R. Reiter, *Knowledge in Action – Logical Foundations for specifying and Implementing Dynamical Systems* (MIT Press, 2001).

[65] M. Thielscher, *Reasoning Robots – The Art and Science of Programming Robotic Agents* (Springer, 2005).

[66] P. Doherty and J. Kvarnström, Temporal action logics, *The Handbook of Knowledge Representation*, eds. V. Lifschitz, F. van Harmelen and F. Porter (Elsevier, 2008).

[67] J. McCarthy and P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence, Vol. 4 (Edinburgh U. Press, 1969), pp. 463–502.

[68] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin and R. B. Scherl, GOLOG: A logic programming language for dynamic domains, *Journal of Logic Programming* **31**(1-3) (1997) 59 – 83.

[69] J. McCarthy, Situations and actions and causal laws, tech. rep., Stanford University, CA. (1963).

[70] E. Davis and L. Morgenstern, A first-order theory of

communication and multi-agent plans, *Journal Logic and Computation* **15**(5) (2005) 701–749.

[71] C. Castelfranchi and R. Falcone, Towards a theory of delegation for agent-based systems, *Robotics and Autonomous Systems* **24** (1998) 141–157.

[72] E. Lorini, N. Troquard, A. Herzig and C. Castelfranchi, Delegation and mental states, *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ACM (2007), p. 153.

[73] H. Hexmoor, C. Castelfranchi and R. Falcone (eds.), *Agent Autonomy*, Multiagent Systems, Artificial Societies, and Simulated Organizations,, Vol. 7 (Springer Verlag, 2003).

[74] H. Hexmoor and D. Kortenkamp, Autonomy control software, *An introductory article and special issue of Journal of Experimental and Theoretical Artificial Intelligence* **12**(2) (2000).

[75] G. Dorais, R. Bonasso, D. Kortenkamp, B. Pell and D. Schreckenghost, Adjustable autonomy for human-centered autonomous systems on Mars, *Proceedings of the Mars Society Conference*, (1998).

[76] J. Bradshaw, M. Sierhuis, A. Acquisti, Y. Gawdiak, R. Jeffers, N. Suri and M. Greaves, Adjustable autonomy and teamwork for the personal satellite assistant, *Proceedings of the IJCAI Workshop on Autonomy, Delegation, and Control: Interacting with Autonomous Agents*, (2001).

[77] H. Hexmoor and B. McLaughlan, Computationally adjustable autonomy, *Journal of Scalable Computing: Practive and Experience* **8**(1) (2007) 41–48.

[78] H. Hexmoor, S. Rahimi and R. Chandran, Delegations guided by trust and autonomy, *Web Intelligence and Agent Systems* **6**(2) (2008) 137–155.

[79] R. Alami, F. Ingrand and S. Qutub, A scheme for coordinating multirobot planning activities and plans execution, *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI)*, (1998).

[80] L. E. Parker, Alliance: An architecture for fault tolerant multi-robot cooperation, *IEEE Transactions on Robotics and Automation* **14**(2) (1998) 220–240.

[81] S. Botelho and R. Alami, M+: A scheme for multi-robot cooperation through negotiated task allocation and achievement, *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (1999).

[82] R. Alami and S. C. Botelho, Plan-based multi-robot cooperation, *Revised Papers from the International Seminar on Advances in Plan-Based Control of Robotic Agents,*, *Lecture Notes in Computer Science* **2466**, (Springer-Verlag, London, UK, 2001), pp. 1–20.

[83] L. E. Parker and F. Tang, Building multi-robot coalitions through automated task solution synthesis, *Proceeding of the IEEE, Special Issue on Multi-Robot Systems* **94**(7) (2006) 1289–1305.

[84] Y. Zhang and L. E. Parker, IQ-ASyMTRe: Forming executable coalitions for tightly-coupled multi-robot tasks, *IEEE Transactions on Robotics* **??** (2012) Early Access Article, `https://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6381528`.

[85] N. Kaldra, D. Ferguson and A. Stentz, Hoplites: A market-based framework for planned tight coordination in multirobot teams, *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (2005), pp. 1170–1177.

[86] K. Decker, TAEMS: A framework for environment centered analysis and design of coordination mechanisms, *Foundations of Distributed Artificial Intelligence*, eds. G. O'Hare and N. Jennings (Wiley Inter-Science, 1996).

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

[88] Y. Zhang and L. E. Parker, Considering inter-task resource constraints in task allocation, *Journal of Autonomous Agents and Multi-Agent Systems* **26** (2013) 389–419.

[89] S. Koenig, P. Keskinocak and C. Tovey, Progress on agent coordination with cooperative auctions, *Proceedings of the AAAI Conference on Artificial Intelligence*, (2010).

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

[91] A. Gerevini and I. Serina, LPG: A planner based on local search for planning graphs, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*, (2002).

[92] S. Edelkamp and M. Helmert, The model checking integrated planning system, *AI Magazine* **22**(3) (2001).

[93] H. Younes and R. Simmons, VHPOP: Versatile heuristic partial order planner, *Journal of Artificial Intelligence Research* **20** (2003).

[94] Y. X. Chen, B. W. Wah and C. W. Hsu, Temporal planning using subgoal partitioning and resolution in SGPlan, *Journal of Artificial Intelligence Research* **26** (2006) 323–369.

[95] V. Vidal and H. Geffner, Branching and pruning: an optimal temporal POCL planner based on constraint programming, *Artificial Intelligence* **170**(3) (2006) 298–335.

[96] J. Kvarnström, TALplanner and other extensions to Temporal Action Logic, PhD thesis, Linköpings universitet (2005).

[97] F. Bacchus and F. Kabanza, Using temporal logics to express search control knowledge for planning, *Artificial Intelligence* **116**(1–2) (2000) 123–191.

[98] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wo and F. Yaman, SHOP2: An HTN planning system, *Journal of Artificial Intelligence Research* **20** (December 2003) 379–404.

[99] R. Brafman and C. Domshlak, From one to many: Planning for loosely coupled multi-agent systems, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, (2008).

[100] W. Van Der Hoek and M. Wooldridge, Tractable multiagent planning for epistemic goals, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, ACM New York, NY, USA (2002), pp. 1167–1174.

[101] C. Guestrin, D. Koller and R. Parr, Multiagent planning

with factored MDPs, *Advances in Neural Information Processing Systems* **2** (1998) 1523–1530.

[102] C. Boutilier, Sequential optimality and coordination in multiagent systems, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, Lawrence Erlbaum Associates (1999), pp. 478–485.

[103] C. Boutilier, Planning, learning and coordination in multiagent decision processes, *Proceedings of the Conference on Theoretical aspects of rationality and knowledge*, Morgan Kaufmann Publishers (1996), pp. 195–210.

[104] P. Gmytrasiewicz and P. Doshi, A framework for sequential planning in multi-agent settings, *Journal of Artificial Intelligence Research* **24** (2005) 49–79.

[105] D. Bernstein, R. Givan, N. Immerman and S. Zilberstein, The complexity of decentralized control of Markov decision processes, *Mathematics of Operations Research* **27**(4) (2002) 819–840.

[106] C. Guestrin, S. Venkataraman and D. Koller, Context-specific multiagent coordination and planning with factored MDPs, *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, (2002), pp. 253–259.

[107] R. Becker, S. Zilberstein, V. Lesser and C. Goldman, Solving transition independent decentralized Markov decision processes, *Journal of Artificial Intelligence Research* **22** (2004) 423–455.

[108] R. Alami, S. Fleury, M. Herrb, F. Ingrand and F. Robert, Multi-robot cooperation in the MARTHA project, *IEEE Robotics & Automation Magazine* **5**(1) (1998) 36–47.

[109] D. D. Corkill, Hierarchical planning in a distributed environment, *Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI)*, (1979).

[110] C. Guestrin and G. Gordon, Distributed planning in hierarchical factored MDPs, *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, (2002).

[111] M. DesJardins, E. Durfee, C. Ortiz and M. Wolverton, A survey of research in distributed, continual planning, *AI Magazine* **20**(4) (1999) 13–22.

[112] M. Desjardins and M. Wolverton, Coordinating planning activity and information flow in a distributed planning system, *Proceedings of the AAAI Fall Symposium on Distributed Continual Planning*, (1998).

[113] M. P. Georgeff, Communication and interaction in multiagent planning, *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, (1983).

[114] J. S. Cox, E. H. Durfee and T. Bartold, A distributed framework for solving the multiagent plan coordination problem, *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (2005).

[115] R. Alami and S. S. da Costa Bothelho, Plan-based multirobot cooperation, *Advances in Plan-Based Control of Robotic Agents*, eds. M. Beetz, J. Hertzberg, M. Ghallab and M. Pollack, *Lecture Notes in Computer Science* **2466** (Springer Berlin Heidelberg, 2002), pp. 1–20.

[116] M. H. Burstein and D. V. McDermott, Issues in the development of human-computer mixed-initiative planning, *Advances in Psychology* **113** (1996) 285–303.

[117] M. M. Veloso, A. M. Mulvehill and M. T. Cox, Rationale-supported mixed-initiative case-based planning, *Proceedings of the National Conference on Artificial Intelligence*, (John Wiley & Sons Ltd, 1997), pp. 1072–1077.

[118] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J.-J. Hsu, A. Jonsson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias *et al.*, Mapgen: Mixed-initiative planning and scheduling for the Mars exploration rover mission, *Intelligent Systems, IEEE* **19**(1) (2004) 8–12.

[119] E. Horvitz, Principles of mixed-initiative user interfaces, *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, ACM (1999), pp. 159–166.

[120] A. Finzi and A. Orlandini, A mixed-initiative approach to human-robot interaction in rescue scenarios, *International Conference on Automated Planning and Scheduling (ICAPS), Printed Notes of Workshop on Mixed-Initiative Planning and Scheduling*, (2005), pp. 36–43.