Auton Agent Multi-Agent Syst
DOI 10.1007/s10458-009-9079-8

A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems

Patrick Doherty · Jonas Kvarnström · Fredrik Heintz

Abstract Research with autonomous unmanned aircraft systems is reaching a new degree of sophistication where targeted missions require complex types of deliberative capability integrated in a practical manner in such systems. Due to these pragmatic constraints, integration is just as important as theoretical and applied work in developing the actual deliberative functionalities. In this article, we present a temporal logic-based task planning and execution monitoring framework and its integration into a fully deployed rotor-based unmanned aircraft system developed in our laboratory. We use a very challenging emergency services application involving body identification and supply delivery as a vehicle for showing the potential use of such a framework in real-world applications. TALplanner, a temporal logic-based task planner, is used to generate mission plans. Building further on the use of TAL (Temporal Action Logic), we show how knowledge gathered from the appropriate sensors during plan execution can be used to create state structures, incrementally building a partial logical model representing the actual development of the system and its environment over time. We then show how formulas in the same logic can be used to specify the desired behavior of the system and its environment and how violations of such formulas can be detected in a timely manner in an execution monitor subsystem. The pervasive use of logic throughout the higher level deliberative layers of the system architecture provides a solid shared declarative semantics that facilitates the transfer of knowledge between different modules.

Keywords Execution monitoring · Planning · Temporal action logic · Reasoning about action and change · Intelligent autonomous systems · Unmanned aircraft systems

P. Doherty · J. Kvarnström (✉) · F. Heintz
Department of Computer and Information Science, Linköpings Universitet, 581 83 Linköping, Sweden
e-mail: jonkv@ida.liu.se

P. Doherty
e-mail: pdy@ida.liu.se

F. Heintz
e-mail: frehe@ida.liu.se

Published online: 21 February 2009

1 Introduction

Now and then, things will go wrong. This is both a fact of life and a fundamental problem in any robotic system intended to operate autonomously or semi-autonomously in the real world. Like humans, robotic systems (or more likely their designers) must be able to take this into account and recover from failures, regardless of whether those failures result from mechanical problems, incorrect assumptions about the world, or interference from other agents.

In this article, we present a temporal logic-based task planning and execution monitoring framework and its integration into a fully deployed rotor-based unmanned aircraft system [15,16] developed in our laboratory. In the spirit of cognitive robotics, we make specific use of Temporal Action Logic (TAL [22]), a logic for reasoning about action and change. This logic has already been used as the semantic basis for a task planner called TALplanner [50], which is used to generate mission plans that are carried out by an execution subsystem. Building further on the use of TAL, we show how knowledge gathered from the appropriate sensors during plan execution can be used by the knowledge processing middleware framework DyKnow [41,43] to create state structures, incrementally building a partial logical model representing the actual development of the system and its environment over time. We then show how formulas in the same logic can be used to specify the desired behavior of the system and its environment and how violations of such formulas can be detected in a timely manner in an execution monitor subsystem which makes use of a progression algorithm for prompt failure detection.

The pervasive use of logic throughout the higher level deliberative layers of the system architecture provides a solid shared declarative semantics that facilitates the transfer of knowledge between different modules. Given a plan specified in TAL, for example, it is possible to automatically extract certain necessary conditions that should be monitored during execution.

In order to drive home the importance of practical integration, we also present the hardware and software system architectures in which the task planning and execution monitoring system is embedded. This is important since the system relies on the use of timely extraction of sensor data and its abstraction into state for monitoring and on motion planners and real-time flight mode capability which are used by the task planner in the generation of mission plans.

Experimentation with the system has been done in the context of a challenging emergency services scenario [26] involving body identification of injured civilians on the ground and logistics missions to deliver medical and food supplies to the injured using several UAVs in a cooperative framework [25]. A combination of real missions flown with our platforms and hardware-in-the-loop simulations has been used to verify the practical feasibility of the proposed systems and techniques.

1.1 Contents

In Sect. 2, we provide some background information about research activities in the UASTech Lab. Sections 3 and 4 describe the unmanned aerial vehicles used in our research focusing on both the software and hardware system architectures. Section 5 describes a challenging emergency services scenario based on a tsunami catastrophe which involves two parts to the mission, body identification and supply delivery. Section 6 introduces Temporal Action Logic, while Sect. 7 discusses the use of TALplanner for generating plans in the UAV domain together with the role of Temporal Action Logic in the planner. Section 8 shows how faults can be detected using an execution monitor based on the use of conditions specified as formulas in TAL. Section 9 builds on the two preceding sections by showing how planning and monitoring can be integrated in a single framework. In Sect. 10 we show how monitor formulas can be generated automatically from a planning domain specification. Section 11 discusses the use of monitor formulas in the presence of inaccurate sensor values. Section 12 contains an empirical evaluation of the execution monitor, and Sect. 13 compares the approach developed in this article to related work. Finally, conclusions and some future directions are presented in Sect. 14.

2 The UASTech lab: autonomous UAS research

The emerging area of intelligent unmanned aerial vehicle (UAV) research has shown rapid development in recent years and offers a great number of research challenges in the area of distributed autonomous robotics systems. Much previous research has focused on low-level control capability with the goal of developing controllers which support the autonomous flight of a UAV from one way-point to another. The most common type of mission scenario involves placing sensor payloads in position for data collection tasks where the data is eventually processed off-line or in real-time by ground personnel. Use of UAVs and mission tasks such as these have become increasingly more important in recent conflict situations and are predicted to play increasingly more important roles in any future conflicts.

Intelligent UAVs will play an equally important role in civil applications. For both military and civil applications, there is a desire to develop more sophisticated UAV platforms where the emphasis is placed on the development of intelligent capabilities and on abilities to interact with human operators and additional robotic platforms. The focus in this research has moved from low-level control towards a combination of low-level and decision-level control integrated in sophisticated software architectures. These, in turn, should also integrate well with larger network-centric based C^4I^2 (Command, Control, Communications, Computers, Intelligence, Interoperability) systems. Such platforms are a prerequisite for supporting the capabilities required for the increasingly more complex mission tasks on the horizon and an ideal testbed for the development and integration of distributed AI technologies.

For a number of years, The Autonomous Unmanned Aircraft Systems Technologies Lab[1] (UASTech Lab) at Linköping University, Sweden, has pursued a long term research endeavour related to the development of future aviation systems in the form of autonomous unmanned aerial vehicles [17,15,16]. The focus has been on both high autonomy (AI related functionality), low level autonomy (traditional control and avionics systems), and their integration in distributed software architectural frameworks [19] which support robust autonomous operation in complex operational environments such as those one would face in catastrophe situations. Some existing application scenarios are traffic monitoring and surveillance, emergency services assistance, photogrammetry and surveying.

Basic and applied research in the project covers a wide range of topics which include the development of a distributed architecture for autonomous unmanned aerial vehicles. In developing the architecture, the larger goals of integration with human operators and other ground and aerial robotics systems in network-centric C^4I^2 infrastructures has been taken into account and influenced the nature of the base architecture. In addition to the software architecture, many AI technologies have been developed such as path planners [61,71,70]

[1] http://www.uastech.com—the UASTech Lab was previously called the UAVTech Lab.

and chronicle recognition and situational awareness techniques [43,44]. In this article, the main focus will be on the use of logics in such higher level deliberative services.

More recently, our research has moved from single platform scenarios to multi-platform scenarios where a combination of UAV platforms with different capabilities are used together with human operators in a mixed-initiative context with adjustable platform autonomy [25]. The application domain we have chosen primarily focuses on emergency services assistance. Such scenarios require a great deal of cooperation among the UAV platforms and between the UAV platforms and human operators.

3 UAV platforms and hardware architecture

The UASTech UAV platform [15] is a slightly modified Yamaha RMAX helicopter (Fig. 1). It has a total length of 3.6 m (including main rotor) and is powered by a 21 hp two-stroke engine with a maximum takeoff weight of 95 kg. Our hardware platform is integrated with the Yamaha platform as shown in Fig. 2. It contains three PC104 embedded computers.

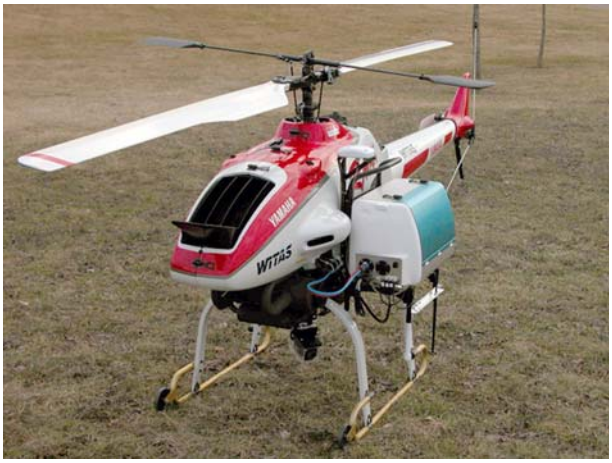


Fig. 1 The UASTech Yamaha RMAX helicopter

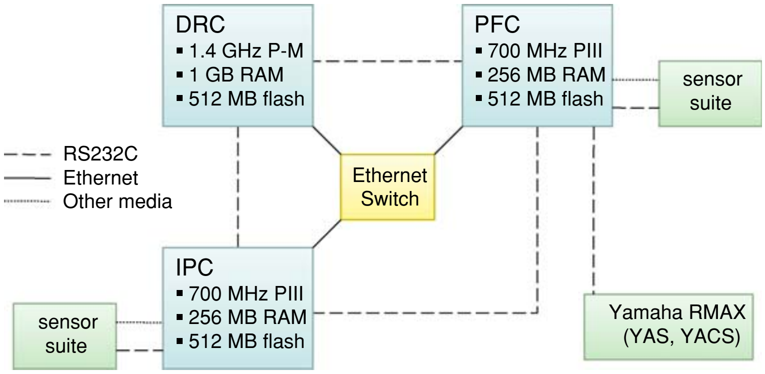


Fig. 2 On-board hardware schematic

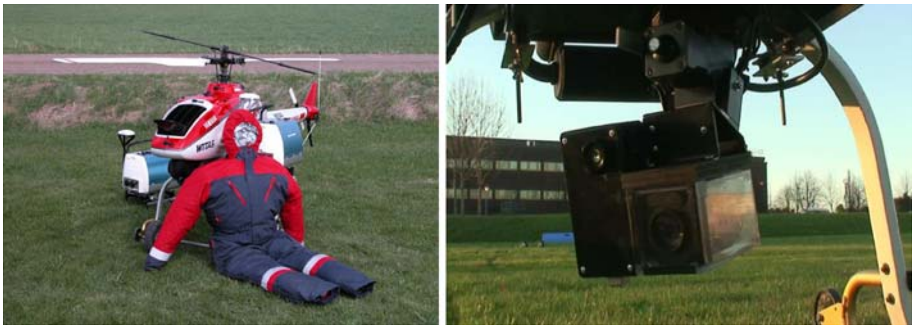


Fig. 3 The UASTech UAV and the on-board camera system mounted on a pan-tilt unit

The primary flight control (PFC) system runs on a PIII (700 MHz), and includes a wireless Ethernet bridge, an RTK GPS receiver, and several additional sensors including a barometric altitude sensor. The PFC is connected to the YAS and YACS,[2] an image processing computer and a computer for deliberative capabilities.

The image processing system (IPC) runs on a second 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 ThermalEye-3600AS miniature infrared camera mounted rigidly on a Pan-Tilt Unit (PTU) as presented in Fig. 3. The video footage from both cameras is recorded at a full frame rate by two MiniDV recorders to allow postprocessing after a flight.

The deliberative/reactive (D/R, DRC) system runs on a third PC104 embedded computer (Pentium-M 1.4 GHz) and executes all high-end autonomous functionality. Network communication between computers is physically realized with serial line RS232C and Ethernet. Ethernet is mainly used for CORBA applications (see below), and remote login and file transfer, while serial lines are used for hard real-time networking.

More recently, we have developed a number of micro aerial vehicles [28,64] for our experimentation with cooperative UAV systems. The intent is to use these together with our RMAX systems for cooperative missions.

4 The software system architecture

A hybrid deliberative/reactive software architecture has been developed for our RMAX UAVs. Conceptually, it is a layered, hierarchical system with deliberative, reactive and control components. Figure 4 presents the functional layer structure of the architecture and emphasizes its reactive-concentric nature. With respect to timing characteristics, the architecture can be divided into two layers: (a) the hard real-time part, which mostly deals with hardware and control laws (also referred to as the Control Kernel) and (b) the non real-time part, which includes deliberative services of the system (also referred to as the High-Level System).[3]

[2] YAS and YACS are acronyms for Yamaha Attitude Sensors and Yamaha Attitude Control System, respectively.

[3] Note that the distinction between the Control Kernel and the High-Level System is conceptual and based mainly on timing characteristics; it does not exclude, for example, placing deliberative services (e.g. prediction) in the Control Kernel.

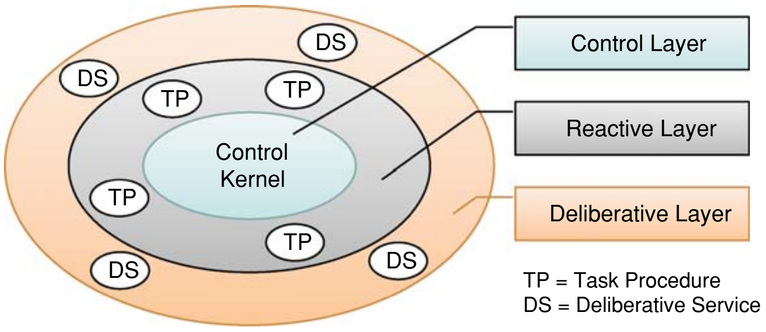


Fig. 4 The functional structure of the architecture

All three computers in our UAV platform (i.e. PFC, IPC and DRC) have both hard and soft real-time components but the processor time is assigned to them in different proportions. On one extreme, the PFC runs mostly hard real-time tasks with only minimum user space applications (e.g. SSH daemon for remote login). On the other extreme, the DRC uses the real-time part only for device drivers and real-time communication. The majority of processor time is spent on running the deliberative services. Among others, the most important ones from the perspective of navigation are the Path Planner, the Task Procedure Execution Module and the Helicopter Server which encapsulates the Control Kernel (CK) of the UAV system.

The CK is a distributed real-time runtime environment and is used for accessing the hardware, implementing continuous control laws, and control mode switching. Moreover, the CK coordinates the real-time communication between all three on-board computers as well as between CKs of other robotic systems. In our case, we perform multi-platform missions with two identical RMAX helicopter platforms. The CK is implemented using C code. This part of the system uses the Real-Time Application Interface (RTAI) [53] which provides industrial-grade real-time operating system functionality. RTAI is a hard real-time extension to a standard Linux kernel (Debian in our case) and has been developed at the Department of Aerospace Engineering of Politecnico di Milano (DIAPM).

The real-time performance is achieved by inserting a module into the Linux kernel space. Since the module takes full control over the processor it is necessary to suspend it in order to let the user space applications run. The standard Linux distribution is a task with lower priority, which is run preemptively and can be interrupted at any time. For that reason a locking mechanism is used when both user- and kernel-space processes communicate through shared memory. It is also important to mention that the CK is self-contained and only the part running on the PFC computer is necessary for maintaining flight capabilities. Such separation enhances safety of the operation of the UAV platform which is especially important in urban environments.

The Control Kernel has a hybrid flavor. Its components contain continuous control laws and mode switching is realized using event-driven hierarchical concurrent state machines (HCSMs) [58]. HCSMs can be represented as state transition diagrams and are similar to statecharts [40]. In our system, tables describing transitions derived from such diagrams are passed to the system in the form of text files and are interpreted by a HCSM interpreter at run-time on each of the on-board computers. Thanks to its compact and efficient implementation, the interpreter runs in the real-time part of the system as a task with high execution rate. It allows coordinating all *functional units* of the control system from the lowest level hardware components (e.g. device drivers) through control laws (e.g. hovering, path following) and communication to the interface used by the Helicopter Server.

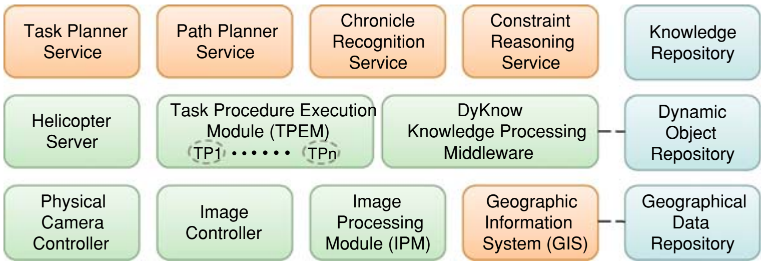
The use of HCSMs also allows implementing complex behaviors consisting of other lower level ones. For instance, landing mode includes control laws steering the helicopter and coordinating camera system/image processing functionalities. When the landing behavior is activated, the CK takes care of searching for a pre-defined pattern with the camera system, feeding the Kalman filter with image processing results which fuses them with the helicopter's inertial measurements. The CK sends appropriate feedback when the landing procedure is finished or it has been aborted. For details see Merz et al. [57].

For achieving best performance, a single non-preemptive real-time task is used which follows a predefined static schedule to run all functional units. Similarly, the real-time communication physically realized using serial lines is statically scheduled with respect to packet sizes and rates of sending. For a detailed description see [56].

The high-level part of the system has reduced timing requirements and is responsible for coordinating the execution of reactive Task Procedures (TPs). A TP is a high-level procedural execution component which provides a computational mechanism for achieving different robotic behaviors by using both deliberative services and traditional control components in a highly distributed and concurrent manner. The control and sensing components of the system are accessible for TPs through the Helicopter Server which in turn uses an interface provided by the Control Kernel. A TP can initiate one of the autonomous control flight modes available in the UAV (e.g. take off, vision-based landing, hovering, dynamic path following or reactive flight modes for interception and tracking). Additionally, TPs can control the payload of the UAV platform which currently consists of video and thermal cameras mounted on a pan-tilt unit in addition to a stereo camera system. TPs can also receive helicopter state delivered by the PFC computer and camera system state delivered by the IPC computer, including image processing results.

The software implementation of the High-Level System is based on CORBA (Common Object Request Broker Architecture), which is often used as middleware for object-based distributed systems. It enables different objects or components to communicate with each other regardless of the programming languages in which they are written, their location on different processors, or the operating systems they are running in. A component can act as a client, a server, or as both. The functional interfaces to components are specified via the use of IDL (Interface Definition Language). The majority of the functionalities which are part of the architecture can be viewed as CORBA objects or collections of objects, where the communication infrastructure is provided by CORBA facilities and other services such as real-time and standard event channels.

This architectural choice provides us with an ideal development environment and versatile run-time system with built-in scalability, modularity, software relocatability on various hardware configurations, performance (real-time event channels and schedulers), and support for plug-and-play software modules. Figure 5 presents some (not all) of the high-level



Task Planner Service
Path Planner Service
Chronicle Recognition Service
Constraint Reasoning Service
Knowledge Repository
Helicopter Server
Task Procedure Execution Module (TPEM)
TP1
TPn
DyKnow Knowledge Processing Middleware
Dynamic Object Repository
Physical Camera Controller
Image Controller
Image Processing Module (IPM)
Geographic Information System (GIS)
Geographical Data Repository

Fig. 5 Some deliberative, reactive and control services

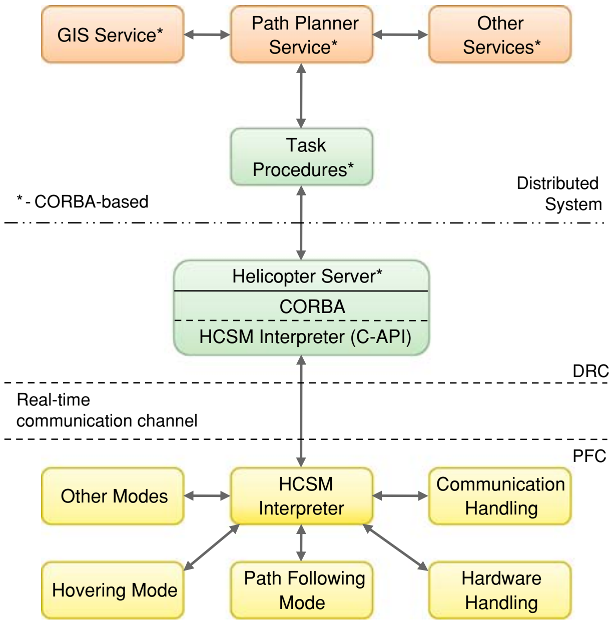


Fig. 6 Navigation subsystem and main software components

services used in the UASTech UAV system. Those services run on the D/R computer and interact with the control system through the Helicopter Server. The Helicopter Server on one side uses CORBA to be accessible by TPs or other components of the system; on the other side it communicates through shared memory with the HCSM based interface running in the real-time part of the DRC software.

Figure 6 depicts the navigation subsystem and main software components.

Currently, we use a number of different techniques for path planning [71,70] which include the use of Probabilistic Roadmaps [46] and Rapidly Exploring Random Trees [48]. For the purposes of this paper, we will simply assume that our UAVs can automatically generate and fly collision free trajectories from specified start and end points in operational environments. The path planner will be called by our task planner in the emergency services scenario considered in this paper.

4.1 DyKnow

A complex autonomous architecture such as the one described above requires timely data, information and knowledge processing on many levels of abstraction. Low-level quantitative sensor data must be processed in multiple steps to eventually generate qualitative data structures which are grounded in the world and can be interpreted as knowledge by higher level deliberative services. DyKnow, a knowledge processing middleware framework, provides this service in the UASTech architecture [41,43].

Conceptually, knowledge processing middleware processes *streams* generated by different components in a distributed system. These streams may be viewed as time-series and for example, may start as streams of observations from sensors or sequences of queries to databases. Knowledge processes combine such streams by computing, synchronizing, filtering and approximating to derive higher level abstractions. For example, an image processing system may be responsible for detecting living beings using streams of images from video and thermal cameras, producing streams of coordinates. DyKnow can also be used as a part of anchoring symbolic representations of objects to sensor data [12], hypothesizing the type of an observed object based on declarative descriptions of normative behaviors [42,44] and recognizing chronicles, collections of events with specific temporal relations [37].

Any service or knowledge process interested in data, information or knowledge, can create or have associated with itself, a declarative specification of quality of service properties such as the maximum permitted delay and strategies for calculating missing values, which together define the properties of the information required by the process.

It is important to realize that knowledge is not static, but is a continually evolving collection of structures which are updated as new information becomes available from sensors and other sources. Therefore, the emphasis is on the continuous and ongoing knowledge derivation process, which can be monitored and influenced at runtime. The same streams of information may be processed differently by different parts of the architecture by tailoring knowledge processes relative to the needs and constraints associated with the tasks at hand. This allows DyKnow to support the integration of existing sensors, databases, reasoning engines and other knowledge processing services.

4.2 Task planning and execution monitoring system overview

The main part of the architecture we will focus on for the remainder of the paper involves those components associated with task planning, execution of task plans and execution monitoring. Figure 7 shows the relevant part of the UAV system architecture associated with these components.

At the top of the center column is the *plan executor* which given a mission request calls the knowledge processing middleware DyKnow to acquire essential information about the current contextual state of the world or the UAV's own internal states. Together with a domain specification and a goal specification related to the current mission, this information is fed to *TALplanner* [21,50,22], a logic-based task planner which outputs a plan that will achieve the designated goals, under the assumption that all actions succeed and no failures occur. Such a plan can also be automatically annotated with global and/or operator-specific conditions to be monitored during execution of the plan by an execution monitor in order to relax the assumption that no failures can occur. Such conditions are expressed as temporal logical formulas and evaluated on-line using formula progression techniques. The execution monitor notifies the command executor when actions do not achieve their desired results and one can then move into a plan repair phase.

The plan executor translates operators in the high-level plan returned by TALplanner into lower level command sequences which are given to the *command executor*. The command executor is responsible for controlling the UAV, either by directly calling the functionality exposed by its lowest level Flight Command Language (FCL) interface or by using Task Procedures through the *Task Procedure Execution Module*.

During plan execution, the command executor adds formulas to be monitored to the *execution monitor*. DyKnow continuously sends information about the development of the world in terms of state sequences to the monitor, which uses a progression algorithm to partially evaluate monitor formulas. If a violation is detected, this is immediately signaled as an event to the command executor, which can suspend the execution of the current plan, invoke an emergency brake command if required, optionally execute an initial recovery action, and

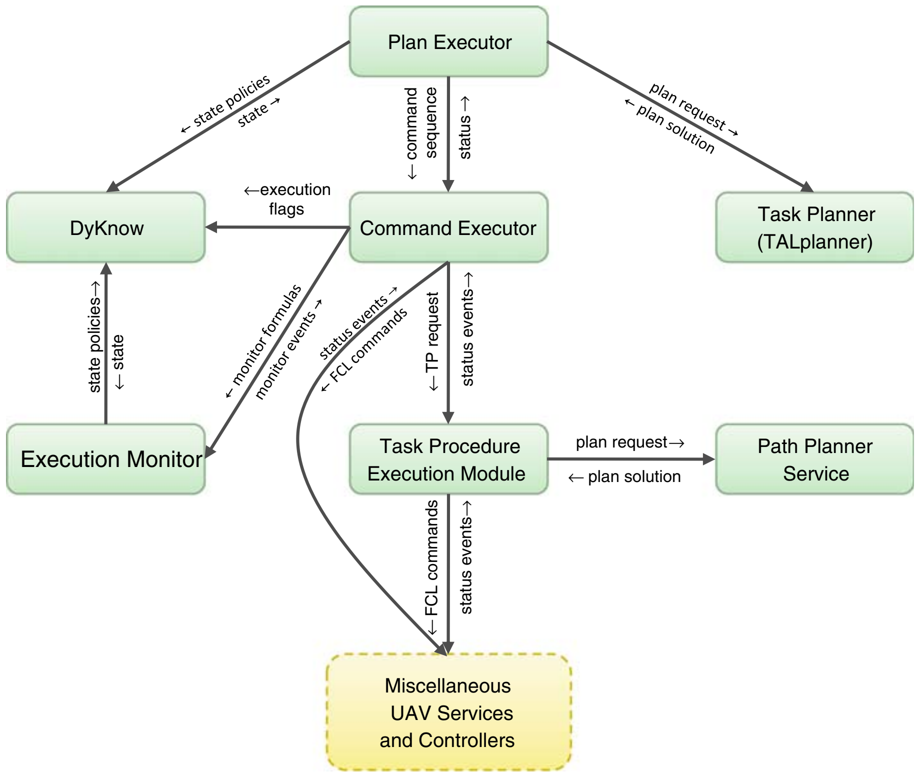


Fig. 7 Task planning and execution monitoring overview

finally signal new status to the plan executor. The plan executor is then responsible for completing the recovery procedure (Sect. 8.4).

The fully integrated system is implemented on our UAVs and can be used onboard for different configurations of the logistics mission described in Leg II of the larger mission. The simulated environments used are in urban areas and quite complex. Plans are generated in the millisecond to seconds range using TALplanner and empirical testing shows that this approach is promising in terms of integrating high-level deliberative capability with lower-level reactive and control functionality.

5 An emergency service scenario

The research methodology used within our group has been very much scenario based, where very challenging scenarios out of reach of current systems are specified and serve as longer term goals to drive both theoretical and applied research. Most importantly, attempts are always made to close the theory/application loop by implementing and integrating results in our UAVs and deploying them for empirical testing at an early stage. One then iterates and continually increases the robustness and functionalities of the targeted components. Currently, we are focusing on an ambitious emergency services scenario which includes the development of cooperative UAV activity. This is the scenario which will be used throughout the rest of the paper and we now describe it.

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

Let us assume for a particular geographic area, one had a shortage of trained helicopter and fixed-wing pilots and/or a shortage of helicopters and other aircraft. Let us also assume that one did have access to a fleet of autonomous unmanned helicopter systems with ground operation facilities. How could such a resource be used in the real-life scenario described?

A pre-requisite for the successful operation of this fleet would be the existence of a multi-agent (UAV platforms, ground operators, etc.) software infrastructure for assisting emergency services in such a catastrophe situation. At the very least, one would require the system to allow mixed initiative interaction with multiple platforms and ground operators in a robust, safe and dependable manner. As far as the individual platforms are concerned, one would require a number of different capabilities, not necessarily shared by each individual platform, but by the fleet in total. These capabilities would include:

– the ability to scan and search for salient entities such as injured humans, building structures or vehicles;
– the ability to monitor or surveil these salient points of interest and continually collect and communicate information back to ground operators and other platforms to keep them situationally aware of current conditions;
– the ability to deliver supplies or resources to these salient points of interest if required. For example, identified injured persons should immediately receive a relief package containing food, water and medical supplies.

Although quite an ambitious set of capabilities, several of them have already been achieved to some extent using our experimental helicopter platforms, although one has a long way to go in terms of an integrated, safe and robust system of systems.

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

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

Leg II In the second part of the scenario, the saliency map generated in Leg I would be used as a basis for generating a logistics plan for several of the UAVs with the appropriate capabilities to deliver food, water and medical supplies to the injured identified in Leg I. This of course would also be done in a cooperative manner among the platforms.

5.1 Mission leg I: body identification

The task of the 1st leg of the mission is to scan a large region with one or more UAVs, identify injured civilians and output a saliency map which can be used by emergency services or other UAVs. Our approach [26,63] uses two video sources (thermal and color) and allows for high rate human detection at larger distances than in the case of using the video sources separately with standard techniques. The high processing rate is essential in case of video collected onboard a UAV in order not to miss potential victims as a UAV flies over them.

A thermal image is analyzed first to find human body sized silhouettes. Corresponding regions in a color image are subjected to a human body classifier which is configured to allow weak classifications. This focus of attention allows for maintaining a body classification at a rate up to 25 Hz. This high processing rate allows for collecting statistics about classified humans and to prune false classifications of the "weak" human body classifier. Detected human bodies are geolocalized on a map which can be used to plan supply delivery. The technique presented has been tested on-board the UASTech helicopter platform and is an important component in our research with autonomous search and rescue missions.

5.1.1 Experimental setup

A series of flight tests were performed in southern Sweden at an emergency services training center used by the Swedish Rescue Services Agency to train fire, police and medical personnel. This area consists of a collection of buildings, roads and even makeshift car and train accidents. This provides an ideal test area for experimenting with traffic surveillance, photogrammetric and surveying scenarios, in addition to scenarios involving emergency services. We have also constructed an accurate 3D model for this area which has proven invaluable in simulation experiments and parts of which have been used in the onboard GIS (Geographic Information System).

Flight tests were performed over varied terrain such as asphalt and gravel roads, grass, trees, water and building roof tops which resulted in a variety of textures in the images. Two UAVs were used over a search area of 290×185 m. A total of eleven bodies (both human and dummies with close to human temperature) were placed in the area. The goal of the mission was to generate a saliency map. The general mission plan is shown in Fig. 8.

Before take-off, one of the UAVs was given an area to scan (dashed line polygon). It then delegated part of the scanning task to another platform, generating sub-plans for itself and the other platform. The mission started with a simultaneous autonomous take-off at positions H_1 and H_2 and the UAVs flew to starting positions S_1 and S_2 for scanning. Throughout the flights, saliency maps were incrementally constructed until the UAVs reached their ending positions E_1 and E_2. The UAVs then returned to their respective take-off positions for a simultaneous landing. The mission took 10 min to complete and each UAV traveled a distance of around 1 km.

5.1.2 Experimental results

The algorithm found all eleven bodies placed in the area. The saliency map generated by one of the helicopters is shown in Fig. 9. The images of identified objects are presented in Fig. 10. Several positions were rejected as they were not observed long enough (i.e. 5 s). Images 7, 9, and 14 present three falsely identified objects.

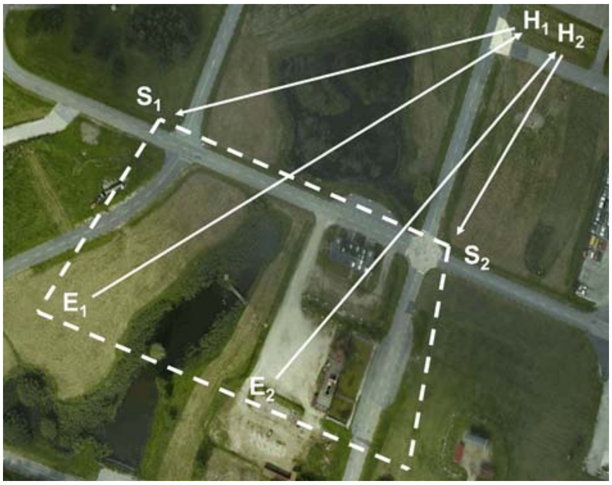


Fig. 8 Mission overview

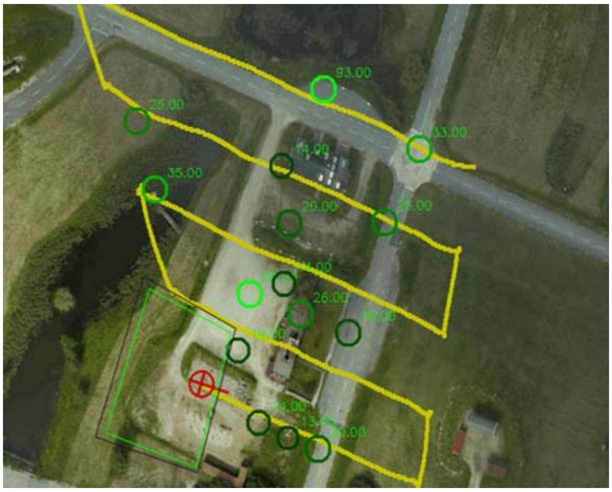


Fig. 9 Flight path and geolocated body positions

5.2 Mission leg II: package delivery

After successful completion of leg I of the mission scenario, we can assume that a saliency map has been generated with geo-located positions of the injured civilians. In the next phase of the mission, the goal is to deliver configurations of medical, food and water supplies to the injured. In order to achieve this leg of the mission, one would require a task planner to plan for logistics, a motion planner to get one or more UAVs to supply and delivery points and an execution monitor to monitor the execution of highly complex plan operators. Each of these functionalities would also have to be tightly integrated in the system. These components have been briefly described in Sects. 4 and 4.2.

This leg of the mission will be used as a running example for the rest of this article. For these logistics missions, we assume the use of one or more UAVs with diverse roles and

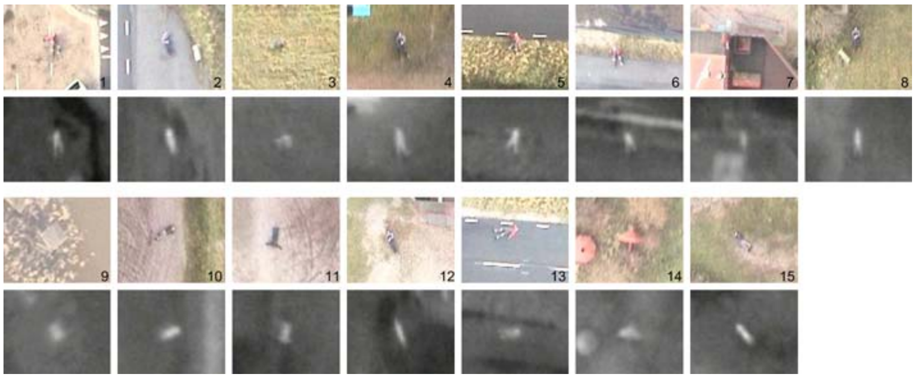


Fig. 10 Images of classified bodies—corresponding thermal images are placed under color images

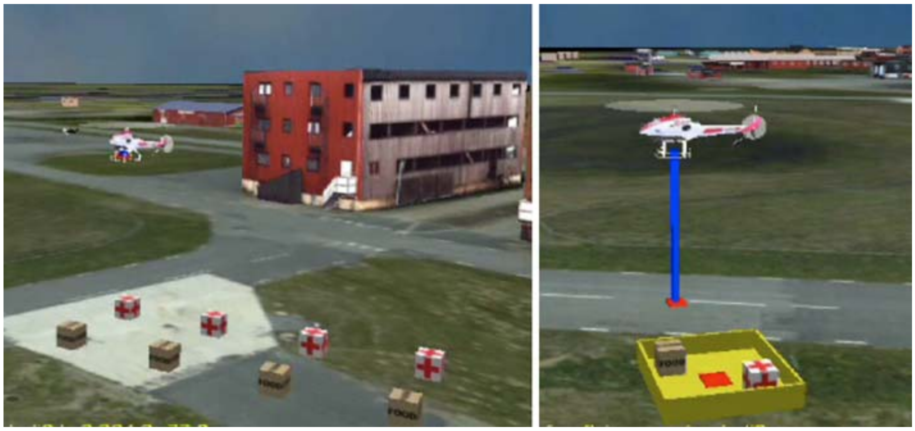


Fig. 11 A supply depot (*left*) and a carrier depot (*right*)

capabilities. Initially, we assume there are n injured body locations, several supply depots and several supply carrier depots (see Fig. 11). The logistics mission is comprised of one or more UAVs transporting boxes containing food and medical supplies between different locations (Fig. 12).

Achieving the goals of such a logistics mission with full autonomy requires the ability to pick up and deliver boxes without human assistance; thus, each UAV has a device for attaching to boxes and carrying them beneath the UAV. The action of picking up a box involves hovering above the box, lowering the device, attaching to the box, and raising the device, after which the box can be transported to its destination. There can also be a number of carriers, each of which is able to carry several boxes. By loading boxes onto such a carrier and then attaching to the carrier, the transportation capacity of a UAV increases manyfold over longer distances. The ultimate mission for the UAVs is to transport the food and medical supplies to their destinations as efficiently as possible using the carriers and boxes at their disposal.

An attachment device consisting of a winch and an electromagnet is under development. In the mean time, the logistics scenario has been implemented and tested in a simulated UAV environment with hardware in-the-loop, where TALplanner generates a detailed mission plan which is then sent to a simulated execution system using the same helicopter flight

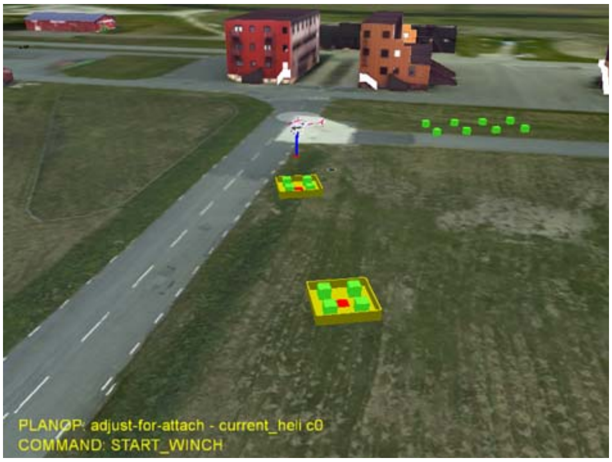


Fig. 12 The UAV logistics simulator

control software as the physical UAV. Each UAV has an execution monitor subsystem which continuously monitors its operation in order to detect and signal any deviations from the declaratively specified intended behavior of the system, allowing the main execution system to initiate the appropriate recovery procedures. Faults can be injected through the simulation system, enabling a large variety of deviations to be tested. Additionally, the simulator makes use of the Open Dynamics Engine,[4] a library for simulating rigid body dynamics, in order to realistically emulate the physics of boxes and carriers. This leads to effects such as boxes bouncing and rolling away from the impact point should they accidentally be dropped, which is also an excellent source of unexpected situations that can be used for validating both the domain model and the execution monitoring system.

6 Background: temporal action logic

This section introduces TAL (Temporal Action Logic), a framework used for reasoning about action and change. More specifically, this section will focus on the use of TAL-C [45,22], one of the more recent members of the TAL family of logics with support for concurrent actions. Though no new results will be presented here, a basic understanding of TAL will be useful when reading other parts of this article. We refer the reader to Doherty and Kvarnström [22] or Doherty et al. [18] for further details.

An agent using TAL is assumed to be interested in one or more reasoning tasks, such as prediction or planning, related to a specific *world* such as the UAV domain. It is assumed that the world is dynamic, in the sense that the various properties or *features* of the world can change over time. TAL-C is an order-sorted logic where each feature, as well as each of its parameters, is specified to take values in a specific *value domain*. The value domain $\text{BOOLEAN} = \{\textbf{true}, \textbf{false}\}$ is always assumed to be present. In addition, the UAV domain could define the two value domains $\text{UAV} = \{\textbf{heli1}, \textbf{heli2}\}$ for UAVs and $\text{BOX} = \{\textbf{bx1}, \textbf{bx2}, \textbf{bx3}, \textbf{bx4}\}$ for boxes to be delivered. The parameterized boolean-valued feature $\mathsf{attached}(uav, box)$ could then represent the fact that *uav* has picked up *box*, while the integer-valued feature $\mathsf{capacity}(uav)$ could be used to model the carrying capacity of *uav*.

4 http://www.ode.org.

TAL offers a modular means of choosing temporal structures depending on the nature of the world being reasoned about and the reasoning abilities of the agent. TAL-C is based on the use of a linear (as opposed to branching) discrete non-negative integer time structure where time 0 corresponds to the initial state in a reasoning problem. The temporal sort is assumed to be interpreted, but can be axiomatized in first-order logic as a subset of Presburger arithmetic, natural numbers with addition [47]. Milliseconds will be used as the primary unit of time throughout this article, where, e.g., the timepoint 4217 is be interpreted as "4.217 seconds after the beginning of the current reasoning problem".

Conceptually, the development of the world over a (possibly infinite) period of time can be viewed in two different ways: As a sequence of *states*, where each state provides a value to each feature (or "state variable") for a single common timepoint, or as a set of *fluents*, where each fluent is a function of time specifying the value of a single feature at each timepoint. The terms "feature" and "fluent" will sometimes be used interchangeably to refer to either a specific property of the world or the function specifying its value over time.

6.1 TAL narratives

TAL is based on the use of narratives specifying background knowledge together with information about a specific reasoning problem. Narratives are initially specified in the narrative description language $\mathcal{L}$(ND), which provides support to a knowledge engineer through a set of high-level macros suitable for a particular task and may vary considerably between TAL logics. The semantics of the language is defined in terms of a translation into first- and second-order logical theories in the language $\mathcal{L}$(FL) which remains essentially unmodified.

The *narrative background specification* contains background knowledge associated with a reasoning domain. *Domain constraint statements* in $\mathcal{L}$(ND) represent facts true in all scenarios associated with a particular reasoning domain, such as the fact that the altitude of a helicopter will always be greater than zero. *Dependency constraint statements* can be used to represent causal theories or assertions which model intricate dependencies describing how and when features change relative to each other. *Action type specifications* specify knowledge about actions, including preconditions and context-dependent effects. Performing an action changes the state of the world according to a set of given rules, which are not necessarily deterministic. For example, the action of tossing a coin can be modeled within the TAL framework, and there will be two possible result states.

The *narrative specification* contains information related to a specific problem instance or reasoning task. *Observation statements* represent observations made by an agent; in the context of planning, this may be used to specify information regarding the initial state. *Action occurrence statements* state which actions occur and provide parameters for those actions.

6.2 The logical base language $\mathcal{L}$(FL)

As noted above, TAL is order-sorted. An $\mathcal{L}$(FL) vocabulary specifies a number of sorts for values $\mathcal{V}_i$, each of which corresponds to a value domain. The sort $\mathcal{V}$ is assumed to be a supersort of all value sorts. There are also a number of sorts $\mathcal{F}_i$ for (reified) features, each one associated with a value sort $dom(\mathcal{F}_i) = \mathcal{V}_j$ for some j. The sort $\mathcal{F}$ is a supersort of all fluent sorts. Finally, there is a sort for actions $\mathcal{A}$ and a temporal sort $\mathcal{T}$.

Variables are typed and range over the values belonging to a specific sort. For convenience, they are usually given the same name as the sort but written in italics, possibly with a prime and/or an index. For example, the variables box, box' and box_3 would be of the sort BOX. Similarly, variables named t are normally temporal variables, and variables named n are normally integer-valued.

$\mathcal{L}$(FL) uses three main predicates. The predicate $Holds : \mathcal{T} \times \mathcal{F} \times \mathcal{V}$ expresses the fact that a feature takes on a certain value at a certain timepoint; for example, $Holds(0, \mathsf{attached}(\mathbf{heli1}, \mathbf{bx3}), \mathbf{true})$ denotes the fact that $\mathsf{attached}(\mathbf{heli1}, \mathbf{bx3})$ has the value **true** at time 0. The predicate $Occlude : \mathcal{T} \times \mathcal{F}$ will be described in the discussion of persistence below. Finally, the predicate $Occurs : \mathcal{T} \times \mathcal{T} \times \mathcal{A}$ specifies what actions occur, and during what intervals of time. The equality predicate is also available, together with the $<$ and $\leq$ relations on the temporal sort $\mathcal{T}$. We sometimes write $\tau \leq \tau' < \tau''$ to denote the conjunction $\tau \leq \tau' \wedge \tau' < \tau''$, and similarly for other combinations of the relation symbols $\leq$ and $<$.

The function $value(\tau, f)$ returns the value of the fluent f at time τ. Formulas in $\mathcal{L}$(FL) are formed using these predicates and functions together with the standard connectives and quantifiers in the usual manner.

6.3 The high-level macro language $\mathcal{L}$(ND)

The following is a small subset of the $\mathcal{L}$(ND) language which is sufficient for the purpose of this article. Fluent formulas provide a convenient means of expressing complex conditions. Fixed fluent formulas provide a temporal context specifying when a fluent formula should hold.

Definition 1 (*Fluent formulas, fixed fluent formulas*) An *elementary fluent formula* has the form $f \hat{=} \omega$ where f is a fluent term of sort $\mathcal{F}_i$ and ω is a value term of sort $dom(\mathcal{F}_i)$. This formula denotes the atemporal fact that the feature f takes on the value ω. A *fluent formula* is an elementary fluent formula or a combination of fluent formulas formed with the standard logical connectives and quantification over values. A *fixed fluent formula* takes the form $[\tau, \tau']\ \alpha$, $(\tau, \tau']\ \alpha$, $[\tau, \tau')\ \alpha$, $(\tau, \tau')\ \alpha$, $[\tau, \infty)\ \alpha$, $(\tau, \infty)\ \alpha$ or $[\tau]\ \alpha$, where α is a fluent formula and τ and τ' are temporal terms. This denotes the fact that the formula α holds at the given timepoint or throughout the given temporal interval.

The elementary fluent formula $f \hat{=}$ **true** ($f \hat{=}$ **false**) can be abbreviated f ($\neg f$). Note that $f = f'$ means that f and f' refer to the same feature, while $f \hat{=} \omega$ denotes the fact that f takes on the value ω at the timepoint specified by the temporal context. The infinity symbol ∞ is not a temporal term but denotes the lack of an upper bound; stating $[\tau, \infty)\ \phi$ is equivalent to stating $\forall t.t \geq \tau \rightarrow [t]\ \phi$.

6.4 Persistence and action effects

In most cases one would like to make the assumption that a feature is *persistent*, only changing values when there is a particular reason, such as an action whose effects explicitly modify the feature. This should be done in a non-monotonic and defeasible manner allowing the incremental addition of new reasons why the feature can change. Several of the fundamental problems in developing logics for reasoning about action and change are related to finding representationally and computationally efficient ways to encode such assumptions without the need to explicitly specify every feature that an action does *not* modify.

The TAL approach uses an occlusion predicate, where $Occlude(\tau, f)$ means that f is allowed to, but does not have to, change values at time τ. Action effects are specified in $\mathcal{L}$(ND) using the R and I reassignment macros. For example, $R([\tau]\ \alpha)$ models an action

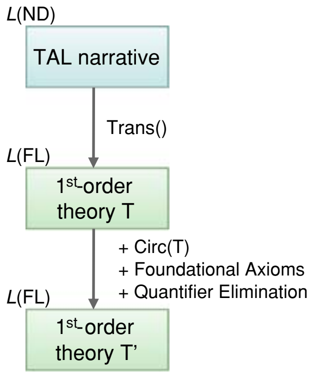


Fig. 13 Reasoning in TAL

effect that causes α to hold at time τ, where α is an arbitrary fluent formula, while $I\,([\tau, \tau'])\ \alpha$ forces α to hold over an interval of time. This is translated into an $\mathcal{L}$(FL) formula where the *Holds* predicate is used to ensure that α holds at the specified timepoint or interval and the *Occlude* predicate is used to ensure that all features occurring in α are occluded, also at the specified timepoint or interval.

A circumscription axiom is used to ensure that features are only occluded when explicitly specified to be occluded. The resulting theory is then conjoined with axioms stating that at any timepoint when a persistent feature is not occluded, it retains its value from the previous timepoint [22,18,14].

Since persistence is not suitable for all features, TAL also supports *durational* features that revert to a default value when not occluded as well as *dynamic* fluents where no persistence or default value assumption is made. Feature types are specified in a fine-grained and flexible manner where the feature type can vary by instance or vary over time. For the remainder of this article, though, all features will be assumed to be persistent.

6.5 Reasoning about TAL narratives

In order to reason about a particular narrative in $\mathcal{L}$(ND), it is first mechanically translated into the base language $\mathcal{L}$(FL) using the *Trans* function as seen in Fig. 13. A circumscription policy is applied to the *Occurs* and *Occlude* predicates in the resulting theory, ensuring that no actions occur except those explicitly specified in action occurrence statements and no fluents are permitted to change except when explicitly specified. A set of foundational axioms are added, including domain closure axioms and unique names axioms where appropriate. Finally, due to certain structural constraints on action type specifications and dependency constraint statements, quantifier elimination techniques can be used to reduce the resulting circumscribed second order theory to a first order theory. [23,24,14].

7 Planning for the UAV domain

When developing the architecture for a system capable of autonomous action execution and goal achievement, one can envision a spectrum of possibilities ranging from each behavior and task being explicitly coded into the system, regardless of complexity, up to the other extreme where the system itself generates complex solutions composed from a set of very primitive low-level actions. With the former end of the spectrum generally leading to more computationally efficient solutions and the latter end generally being far more flexible in the event of new and potentially unexpected tasks being tackled, the proper choice is usually somewhere between the two extremes; in fact, several different points along the spectrum might be appropriate for use in different parts of a complex system. This is also the case for our UAV system, which provides a set of high-level actions such as "take off" and "fly to point A" but also makes use of planning techniques to compose such actions into plans satisfying a set of declaratively specified goals. The transportation of medical supplies is only one of many possible scenarios that can be modeled in this manner.

The planner used for this purpose is TALplanner [20,21,51,50], a forward-chaining planner where planning domains and problem instances are specified using a version of TAL-C extended with new macros for plan operators, resource constraints, goal specifications, and other issues specific to the planning task. In addition to providing a declarative first-order semantics for planning domains, thereby serving as a specification for the proper behavior of the planning algorithm, TAL is also used to specify a set of temporal control formulas acting as constraints on the set of valid plans. Such formulas can be used for specifying temporally extended goals that must be satisfied over the (predicted) execution of a plan. In addition, they can be used to constrain the forward-chaining search procedure, guiding the planner towards those parts of the search space that are more likely to contain plans of high quality in terms of flight time, delivery delays or other quality measures.

In the remainder of this section, we will first show how TAL can be used for modeling the UAV logistics scenario (Sect. 7.1). We then discuss the use of control formulas in TALplanner and how they constrain the set of valid plans (Sect. 7.2). Due to the fine granularity with which operators have been modeled, the typical plan length for a small example with four boxes, one carrier and one UAV is approximately 150 to 250 actions, depending on the initial state and the goal. Such plans can typically be generated in less than one second on a 1.8 GHz Pentium 4 machine. Most likely, some optimizations would be possible if necessary: The planning domain definition has currently been written for readability and ease of modification, not for performance.

See Kvarnström and Doherty [51] or Kvarnström [50] for further details regarding TALplanner.

7.1 Modeling the UAV logistics scenario in TAL

Though many traditional benchmark domains for planning are quite simple, and could be described and formalized in their entirety on a page or two, this is mainly due to two facts: First, many domains are designed to illustrate a specific point; second, a large proportion of the domains were defined at a time when planners could not be expected to handle more complex and detailed domain definitions due to limitations in computational capacity as well as in planning algorithms. The UAV domain discussed here, on the other hand, is intended to be used in a complex real world application where topics such as exact locations, distances and timing (as opposed to symbolic positions and unit timing for each action) are essential and cannot be abstracted away without severely compromising the average quality of a plan, or even the likelihood of it being executable at all. Unfortunately, this means that the complete $\mathcal{L}$(ND) domain description with all associated operator specifications and control formulas is too large to be presented here in its entirety; instead, a number of representative examples will be used.

In terms of value domains, we first require a set of domains to represent objects which have locations. Thus, the top level domain LOCATABLE has the subdomains UAV and CARRYABLE, the latter of which has the subtypes BOX and CARRIER. Additionally, it may not be possible to place carriers at arbitrary positions due to terrain constraints, buildings, and other types of obstacles. For simplicity, the domain CARRIER- POSITION represents intermediate locations where carriers may be placed when loading and unloading boxes. This may eventually be augmented with a method for querying the onboard GIS to dynamically find a suitable location for carrier placement; however, the requirement of predictability in a UAV deployment is likely to make the use of predefined carrier positions a legal necessity in many cases.

Each LOCATABLE object has a position which may vary over time. The x coordinate of a LOCATABLE is represented by the xpos feature, and the y coordinate by the ypos feature, taking values from a finite domain FP of fixed-point numbers (that is, numbers with a fixed number of decimals).[5] The current altitude of each UAV is modeled using the altitude fluent, also taking values from the domain FP. We appeal to the use of *semantic attachment* [68] techniques in the implementation of TAL and TALplanner by liberal use and invocation of built in mathematical functions and other functions associated with finite integer and fixed-point value domains.

Unlike some benchmark planning domains, where (for example) an unlimited number of boxes can be loaded into a single vehicle, our formalization of the UAV domain must comply with the physics of the real world. Consequently, each UAV has a limited carrying capacity, and each carrier has limited space available for boxes. Modeling this is not difficult, but a detailed model of issues involving weights and shapes would lead to unnecessary complexity in an area which is outside the focus of this article. Instead, a simpler model is used, where all boxes are assumed to be of similar shape, carrier capacities are modeled in terms of the number of boxes that will fit ($\text{carrier-capacity}(carrier) : \text{FP}$), and UAV capacities are modeled in terms of the number of boxes that it can carry ($\text{uav-capacity}(uav) : \text{FP}$).

It should also be mentioned that the UAV is only permitted to place boxes on specific areas on each carrier; boxes placed elsewhere could block the electromagnet from attaching to the carrier, or could fall off when the carrier is lifted. For this reason, the $\text{on-carrier}(box, carrier)$ fluent is not boolean, but has three values representing the cases where the box is definitely not on the carrier, definitely correctly placed on the carrier, and perhaps blocking the carrier, respectively. Correctly deducing the third case also entails modeling the size of each carryable and the minimum safety distances between different types of carryables; this has been done but will not be further discussed here.

The boolean feature $\text{attached}(uav, carryable)$ represents the fact that a certain UAV has attached its electromagnet to a certain carryable. Finally, a number of features are used for modeling abstract conditions such as whether the UAV is prepared and ready to fly, the most prominent one being $\text{state}(uav)$ taking values from the domain UAVSTATE = {**unprepared**, **ready-to-fly**, **ready-to-attach**, **ready-to-detach**}.

7.1.1 Operators

The UAV system provides a varied and complex set of functionalities that can be exposed to the planner as operators. Here, we will focus on the functionality that is the most relevant for the logistics missions used in this article: Flying to different locations, attaching and detaching carryable objects such as boxes and carriers, and (since we want a reasonably fine-grained model of the domain, in order to support fine-grained recovery) various forms of preparatory actions such as adjusting the altitude before attaching a box.

[5] Infinite domains are currently not allowed in TAL, and floating point numbers have a semantics which is considerably more difficult to formalize in logic. Since one can choose an arbitrary precision for a domain of fixed-point numbers, this is not a problem in practice.

Even if the functionality exposed by the lower layers of the UAV architecture were seen as fixed and unalterable, there is still a great deal of flexibility in determining how this should be modeled in the planner: Even if there is a single parameterized *fly* functionality, for example, one may still choose to model this as several different operators for the purposes of planning if this has advantages in terms of readability. This is often the case for a planner such as TALplanner, which supports operator-specific control formulas that might only be applied in certain cases. To be more concrete, five different *fly* operators are used in the TAL domain specification created for the UAV domain: fly-empty-to-box, fly-empty-to-carrier, fly-box-to-carrier, fly-box-to-goal, and fly-carrier. Each of these operators has its own associated set of control formulas, because the action of flying without cargo to pick up a box is appropriate in different circumstances than flying with a box to place it on a carrier.

In addition to the flight operators, a UAV can also adjust-for-attach and then either attach-box or attach-carrier. After a subsequent climb-for-flying-with action, which reels in the winch and climbs to the standard flight altitude, it can fly the carryable to another location, adjust-for-detach, and either detach-box or detach-carrier depending on the type of carryable. After finishing with a climb-for-flying-empty action, the UAV is free to pursue other goals.

For all of these operators, there are a number of parameters, not all of which are relevant for the purpose of discussing the integration between planning and execution monitoring. A couple of examples are in order, though: climb-for-flying-empty(*uav*) requires a UAV as its only parameter. The operator fly-empty-to-carrier(*uav*, *fromx*, *fromy*, *carrier*, *tox*, *toy*) is somewhat more complex, requiring a UAV and its coordinates, plus a destination carrier and its coordinates, as parameters. (Note that if one models an area of 10000 m square at a resolution of 1 centimeter, each coordinate has 10^6 possible values, and even with only a single UAV and a single carrier, the operator has 10^{24} ground instances. Obviously, TALplanner does not generate all ground instances of an operator, as some planners do.)

The exact effects and preconditions of these operators cannot be listed here in their entirety; again, they are not relevant for the purposes of the article. However, we will show one of the simplest operators in the syntax used by TALplanner together with its translation into $\mathcal{L}$(ND) and $\mathcal{L}$(FL).

```
#operator adjust-for-attach(uav, carryable, x, y)
:at start, end
:cost 500
:timeconstraint 0 <= end - start <= 5*UNITS_PER_SECOND
:precond [start] near(uav, carryable, MAX_ATTACH_DIST)
         & !hasCargo(uav)
         & xpos(carryable) == x
         & ypos(carryable) == y
:effects [end] state(uav):= ready-to-attach
```

The operator is named adjust-for-attach and takes four parameters, two of which (x and y) are only used in operator-specific control formulas which are omitted here. The start and end timepoints *start* and *end* can also be seen as implicit parameters and are instantiated by the planner as required during the search process. The :cost clause specifies a cost for this action, which is used if an optimizing search strategy is applied by the planner. Here, the cost is constant, but it can also depend on the arguments of the operator. The :timeconstraint clause provides a constraint on the timing of the operator, which may not be completely determined in advance. The :precondition clause makes use of several feature macros (features defined in terms of logic formulas), the meaning of which should be apparent from their usage. For example, $\mathsf{hasCargo}(uav)$ is defined to hold iff $\exists carryable.\mathsf{attached}(uav, carryable)$. The operator is applicable iff the UAV is sufficiently close to the carryable that should be attached and the UAV is not currently carrying any cargo; the conditions on x and y serve to bind these variables for use in control formulas. Finally, the only effect of this operator is that at the end, the state of the UAV is **ready-to-attach**.

The TAL-C action type specification corresponding to this would be defined as follows. Cost is omitted since this concept is only used during the planning phase and is not part of TAL. Free variables are assumed to be universally quantified.

$$\begin{aligned}
&[start, end]\ \mathsf{adjust\text{-}for\text{-}attach}(uav, carryable, x, y) \rightarrow\\
&0 \leq end - start \leq 5 \cdot \mathbf{UNITS\_PER\_SECOND} \wedge\\
&([start]\ \mathsf{near}(uav, carryable, \mathbf{MAX\_ATTACH\_DIST}) \wedge\\
&\ \neg\mathsf{hasCargo}(uav) \wedge \mathsf{xpos}(carryable) \hat{=} x \wedge \mathsf{ypos}(carryable) \hat{=} y \rightarrow\\
&\ R([end]\ \mathsf{state}(uav) \hat{=} \mathbf{ready\text{-}to\text{-}attach}))
\end{aligned}$$

This would be translated into the following $\mathcal{L}(\mathrm{FL})$ action type specification:

$$\begin{aligned}
&Occurs(start, end, \mathsf{adjust\text{-}for\text{-}attach}(uav, carryable, x, y)) \rightarrow\\
&0 \leq end - start \leq 5 \cdot \mathbf{UNITS\_PER\_SECOND} \wedge\\
&(Holds(start, \mathsf{near}(uav, carryable, \mathbf{MAX\_ATTACH\_DIST}), \mathbf{true}) \wedge\\
&\ \neg Holds(start, \mathsf{hasCargo}(uav), \mathbf{true}) \wedge\\
&\ Holds(start, \mathsf{xpos}(carryable), x) \wedge Holds(start, \mathsf{ypos}(carryable), y) \rightarrow\\
&\ Holds(end, \mathsf{state}(uav), \mathbf{ready\text{-}to\text{-}attach}) \wedge\\
&\ Occlude(end, \mathsf{state}(uav)))
\end{aligned}$$

7.2 Control formulas in TALplanner

Given the operators described above together with an initial state and a set of formulas that must be satisfied in any goal state, the task of the planner is to search for a valid and executable plan that ends in a goal state. If we (for the sake of discussion) temporarily restrict ourselves to finding sequential plans, applying a forward-chaining algorithm to this task entails a search space where the root node is the empty action sequence and where the children of any node n are exactly those generated by appending one more applicable action to the end of the action sequence associated with n.

Clearly, each node n associated with an action sequence p can also be viewed as corresponding to a finite state sequence $[s_0, s_1, \ldots, s_m]$—the sequence that would be generated by executing p starting in the initial state. This state sequence, in turn, can be seen as corresponding to a TAL interpretation. This leads us directly to the possibility of specifying constraints on a plan in terms of TAL formulas to be satisfied by the corresponding interpretation.

Before going into further detail, a small example will be presented.

Example 1 (*Global control formula*) In the UAV Domain, a box should only be moved if the goal requires it to be elsewhere, or if it is blocking some other action. The first condition holds if the box is farther away from its ideal goal coordinates than a given (possibly context-specific) threshold. One cannot require the box to be *exactly* at a given set of coordinates: If a failure is detected the system may need to replan, and one does not want to move boxes again merely because they may be centimeters away from an ideal goal location. The second condition might hold if the box is too close to a carrier position according to a specified safety margin. These conditions are simplified and modularized using several feature macros defined in terms of basic features: close-enough-to-goal(box), need-to-move-temporarily(box), and is-at$(locatable, x, y)$.

$$\begin{aligned}&\forall t, box, x, y.\\&\quad [t]\ \textsf{is-at}(box, x, y) \rightarrow [t+1]\ \textsf{is-at}(box, x, y) \vee\\&\quad [t]\ \neg\textsf{close-enough-to-goal}(box) \vee\\&\quad [t]\ \textsf{need-to-move-temporarily}(box)\end{aligned}$$

□

Some control formulas are always satisfied by inaction; for example, the empty plan generates a state sequence where no boxes are moved, which will satisfy the formula above. Formulas may also require certain changes to take place, however. For example, one could state that if a UAV takes off, it must begin flying to a destination within 30 s; otherwise it is wasting fuel and might as well have waited on the ground.

If fluents are assumed to be persistent (not change values) after the final action in a plan, then a plan consisting of a single takeoff action will permit the conclusion that the UAV takes off and then remains stationary indefinitely, which violates the control formula. This is obviously wrong, since the only reason why the UAV remains stationary is that the plan search algorithm has not yet added a flight action. Only after the planner has actually added actions predicted to take 30 s or more, without adding a flight action, should the control formula be considered to be violated.

This is achieved by considering the state sequence $[s_0, s_1, \ldots, s_m]$ associated with an intermediate node in the search tree to be a *partial* state sequence, a prefix of the final sequence that will eventually be generated. Similarly, the TAL interpretation corresponding to an intermediate node is viewed as a partial interpretation $\mathcal{I}$, where fluent values up to the time of state s_m are completely determined, after which they are assumed to be completely unknown. If ϕ is a control formula and $\mathcal{I} \models \neg\phi$, then both this node and any descendant must necessarily violate the control formula (because descendant nodes can only add new information after s_m which does not retract any previous conclusions), and the planner can reject the node and backtrack. How to test this efficiently is discussed in Kvarnström [49,50].

In addition to global control formulas, TALplanner also permits the use of operator-specific control. Such formulas are similar to preconditions, but whereas preconditions are intended to model constraints on when it is "physically" possible to use an operator, operator-specific control is intended to model constraints on when using the operator is recommended.

Example 2 (*Operator-specific control*) Suppose one wants to express the constraint that a UAV should not prepare for attaching to a carrier where there are potentially misplaced boxes, because such a carrier may not be correctly balanced. This control formula can be declared locally in the adjust-for-attach operator, which also gives it access to operator parameters. Specifically, the *carryable* parameter indicates the relevant carryable, which may or may not be a carrier. This leads to the following conditionalized control formula, which is violated by any action preparing to attach to a carrier where there is a box which is closer than the designated safety distance and is not correctly placed:

$$\begin{aligned}&[start]\ \forall carrier, box.\\&\quad carrier = carryable \wedge \textsf{near}(box, carrier, \textsf{safetyDistance}(box, carrier)) \rightarrow\\&\quad \textsf{on-carrier}(box, carrier) \hat{=} \mathbf{correctly\_placed}\end{aligned}$$

□

8 Execution monitoring

Classical planners are built on the fundamental assumption that the only agent causing change in the environment is the planner itself, or rather, the system or systems that will eventually execute the plan that it generates. Furthermore, they assume that all information provided to the planner as part of the initial state and the operator specifications is accurate. Though this may in some cases be a reasonable approximation of reality, it is more often manifestly untrue: Numerous other agents might manipulate the environment of an autonomous system in ways that may prevent the successful execution of a plan, and actions can sometimes fail to have the effects that were modeled in a planning domain specification regardless of the effort spent modeling all possible contingencies. Consequently, robust performance in a noisy environment requires some form of supervision, where the execution of a plan is constantly monitored in order to detect any discrepancies and recover from potential or actual failures. For example, a UAV might accidentally drop its cargo; thus, it must monitor the condition that if a box is attached, it must remain attached until the UAV reaches its intended destination. This is an example of a *safety constraint*, a condition that must be maintained during the execution of an action or across the execution of multiple actions. The carrier can also be too heavy, which means that one must be able to detect takeoff failures where the UAV fails to gain sufficient altitude. This can be called a *progress constraint*: Instead of maintaining a condition, a condition must be achieved within a certain period of time.

The requirement for monitoring leads to the question of what conditions should be monitored, and how such conditions should be specified. Clearly, there are certain contingencies that would best be monitored by the low-level implementation of an operation or behavior, but universal use of this principle would lead to excessively complex action implementations with duplication of failure detection functionalities and a lack of modularity. As an alternative, the monitoring of failures and the recovery from unintentional situations could be separated from the execution of actions and plans and lifted into a higher level *execution monitor* [9,13,30,31,35,8], where the constraints to be monitored should preferably be specified in an expressive declarative formalism. If a constraint is violated, the execution system should be signaled, after which the UAV can react and attempt to recover from the failure. This is the approach taken in this article.

Our system for monitoring the correct execution of a plan is based on an intuition similar to that underlying the temporal control formulas used in TALplanner. As a plan is being executed, information about the surrounding environment is sampled at a given frequency. Each new sampling point generates a new state which provides information about all fluents used by the current monitor formulas, thereby providing information about the *actual* state of the world at that particular point in time, as opposed to what could be *predicted* from the domain specification. Concatenating all states generated by this sampling process yields a state sequence that corresponds to a partial logical interpretation, where "past" and "present" states are completely specified whereas "future" states are completely undefined (Sect. 8.1).

Given that both actual and predicted states are available, one obvious approach to monitoring would be to simply compare these states and signal a violation as soon as a discrepancy is found. Unfortunately, the trivial application of this approach is not sufficient, because not all discrepancies are fatal: If the altitude was predicted to be 5 m and the current measurement turns out to be 4.984 m, then one most likely does not have to abort the mission. Additionally, some information about the environment might be expensive or difficult to sense, in which case the operator or domain designer should be given more control over when and where such information is used, rather than forcing the system to gather this information continuously in order to provide sufficient information for state generation. Finally, the richer the domain model is, the more the planner can predict about the development of the world; this should not necessarily lead to all those conditions being monitored, if they are not relevant to the correct execution of a plan. Determining which of these predictions are truly important for goal achievement and which are less relevant, and weighing this importance against the difficulty or expense involved in sensing, is best done by a human.

For these reasons, most conditions to be monitored are explicitly specified using a variation of Temporal Action Logic (Sect. 8.2), though many conditions *can* be automatically generated within the same framework if so desired (Sect. 10). For example, a very simple monitor formula might monitor the constraint that at any timepoint, $\forall uav.\mathsf{altitude}(uav) \leq 50$. Through the use of logic, conditions are given an expressive formal declarative semantics where both safety and progress conditions can be defined and monitored.

As each new sensed state arrives into the system, the current set of monitor formulas is tested against the incrementally constructed partial logical model using a progression algorithm (Sect. 8.3), and any violation is reported to the execution system which can take the appropriate action (Sect. 8.4).

8.1 State generation

Ideally, monitor conditions should be tested relative to the actual development of the real world over time. In practice, sensors are not completely accurate, there are limits to the frequency with which they can be sampled, and readings take time to propagate through a distributed system which may consist of multiple UAV platforms together with ground stations and associated hardware and software. The UAV architecture therefore uses the knowledge processing middleware framework DyKnow to gather data from distributed physical and virtual sensors. Such data can also serve as the basis for further computations and data fusion processes which feed higher-level qualitative and quantitative information back into DyKnow.

DyKnow also provides services for data synchronization, generating a best approximation of the state at a given point in time using the information that has propagated through the distributed system so far. By asking about the state n milliseconds ago rather than the state at the current time, one can increase the probability that most sensor data for the given timepoint will have had enough time to arrive in DyKnow. Increasing n increases the accuracy of the state at the cost of lengthening the delay before a constraint violation can be detected; however, the potential for communication failures in any distributed system (which may involve multiple UAVs as well as ground robots) means that a complete correctness guarantee can never be given. This is also the case due to sensor inaccuracy. Consequently, monitor conditions must be written to be robust against minor inaccuracies in the state sequence, as will be seen in Sect. 11.

8.2 Execution monitor formulas

Having decided that the conditions to be tested by the execution monitor should be specified in the form of logic formulas, we still retain a great deal of freedom in choosing exactly which logic should be used. However, there are several important considerations that can be used for guidance.

First, the logic must be able to express conditions over time. In restricted cases, the conditions to be tested could consist of simple state constraints specifying requirements that must hold in each individual state throughout the execution of a plan. In the general case, though, one would need the ability to specify conditions *across* states, such as the fact that when a UAV has attached to a box, the box must remain attached until released.

Second, the logic should be able to express metric constraints on time, in order to provide support for conditions such as the UAV succeeding in attaching a box within at most 10 s.

These are the most important constraints on expressivity, and had these been the only constraints on our system, it would have been possible to use an unmodified version of TAL-C to express monitor formulas. After all, TAL-C has been used to great success in specifying control formulas for the planning phase, and has support for temporally extended constraints as well as metric time. However, there is a third and more pragmatic constraint that must also be satisfied: It must be possible to test each monitor formula *incrementally* and *efficiently* as each new state arrives from DyKnow into the execution monitor, because violations must be detected as early as possible, and there may be a large number of monitor formulas to be tested using the limited computational capabilities of the onboard computers on a UAV.

For TALplanner, the main strategy for verifying that control formulas are not violated is based on the use of plain formula evaluation in a partial interpretation. The efficiency of this strategy is predicated on the use of an extensive pre-planning analysis phase where the control formulas specified for a given domain are analyzed relative to the operators available to the planner [49]. For the execution monitoring system, the prerequisites for the formula analysis phase are not satisfied: The very reason for the existence of the system is the potential for unforeseen and unpredictable events and failures, rendering the analysis relative to specific operators specifying "what could happen" ineffective. For this reason, a formula progression algorithm would be more appropriate for this application, and such algorithms are more easily applied to formulas in a syntax such as that used in modal tense logics such as LTL (Linear Temporal Logic [29]) and MITL (Metric Interval Temporal Logic [1,2]).

Note again the wording above: Progression is more easily applied to formulas in a *syntax* such as that used in modal tense logics. In fact, there is no requirement for actually using such a logic; instead, it is possible to create a new variation of the $\mathcal{L}$(ND) surface language for TAL-C containing operators similar to those in MITL together with a translation into the same base logic $\mathcal{L}$(FL). Doing this has the clear advantage of providing a common semantic ground for planning and execution monitoring, regardless of the surface syntax of a formula.[6] It should be noted that the ability to adapt the surface language to specific applications is in fact one of the main reasons behind the use of two different languages in TAL; the $\mathcal{L}$(ND) language has already been adapted in different ways for planning [50,51], object-oriented modeling [38], and other tasks.

8.2.1 Monitor formulas in TAL

Three new *tense operators* have been introduced into $\mathcal{L}$(ND) for use in formula progression: U(until), $\Diamond$ (eventually), and $\Box$ (always). Note that like all expressions in $\mathcal{L}$(ND), these operators are macros on top of the first order base language $\mathcal{L}$(FL). We also introduce the concept of a *monitor formula* in TAL.

Definition 2 (*Monitor formula*) A *monitor formula* is one of the following:

- $\tau \leq \tau'$, $\tau < \tau'$, or $\tau = \tau'$, where τ and τ' are temporal terms,
- $\omega = \omega'$, where ω and ω' are value terms,
- a fluent formula,

[6] The use of tense operators in TAL was in fact first introduced in TALplanner, which provides both the standard TAL syntax and a tense logic syntax for its control formulas.

- $\phi \, \mathsf{U}_{[\tau,\tau']} \, \psi$, where ϕ and ψ are monitor formulas and τ and τ' are temporal terms,
- $\Diamond_{[\tau,\tau']} \, \phi$, where ϕ is a monitor formula and τ and τ' are temporal terms,
- $\Box_{[\tau,\tau']} \, \phi$, where ϕ is a monitor formula and τ and τ' are temporal terms, and
- a combination of monitor formulas using the standard logical connectives and quantification over values.

The shorthand notation $\phi \, \mathsf{U} \, \psi \equiv \phi \, \mathsf{U}_{[0,\infty)} \, \psi$, $\Diamond \, \phi \equiv \Diamond_{[0,\infty)} \, \phi$, and $\Box \, \phi \equiv \Box_{[0,\infty)} \, \phi$ is also permitted in monitor formulas.

Whereas other logic formulas in $\mathcal{L}$(ND) use absolute time (as in the fixed fluent formula $[\tau] \; f \hat{=} v$), monitor formulas use relative time, where each formula is evaluated relative to a "current" timepoint. The semantics of these formulas will be defined in terms of a translation into $\mathcal{L}$(FL) satisfying the following conditions:

- The formula $\phi \, \mathsf{U}_{[\tau,\tau']} \, \psi$ ("until") holds at time t iff ψ holds at some state with time $t' \in [t+\tau, t+\tau']$ and ϕ holds until then (at all states in $[t, t')$, which may be an empty interval).
- The formula $\Diamond_{[\tau,\tau']} \, \phi$ ("eventually") is equivalent to $\texttt{true} \, \mathsf{U}_{[\tau,\tau']} \, \phi$ and holds at t iff ϕ holds in some state with time $t' \in [t+\tau, t+\tau']$.
- The formula $\Box_{[\tau,\tau']} \, \phi$ is equivalent to $\neg \Diamond_{[\tau,\tau']} \neg\phi$ and holds at t iff ϕ holds in all states with time $t' \in [t+\tau, t+\tau']$.

Definition 3 (*Translation of monitor formulas*) Let $\overline{\tau}$ be a temporal term and γ be a monitor formula intended to be evaluated at $\overline{\tau}$. Then, the following procedure returns an equivalent formula in $\mathcal{L}$(ND) without tense operators.

$$\begin{array}{ll}
\mathsf{TransMonitor}(\overline{\tau}, \mathcal{Q}x.\phi) & \stackrel{\text{def}}{=} \mathcal{Q}x.\mathsf{TransMonitor}(\overline{\tau}, \phi), \text{ where } \mathcal{Q} \text{ is a quantifier} \\
\mathsf{TransMonitor}(\overline{\tau}, \phi \otimes \psi) & \stackrel{\text{def}}{=} \mathsf{TransMonitor}(\overline{\tau}, \phi) \otimes \mathsf{TransMonitor}(\overline{\tau}, \psi), \text{ where } \otimes \text{ is} \\
 & \quad \text{a binary connective} \\
\mathsf{TransMonitor}(\overline{\tau}, \neg\phi) & \stackrel{\text{def}}{=} \neg\mathsf{TransMonitor}(\overline{\tau}, \phi) \\
\mathsf{TransMonitor}(\overline{\tau}, f \hat{=} v) & \stackrel{\text{def}}{=} [\overline{\tau}] \; f \hat{=} v \\
\mathsf{TransMonitor}(\overline{\tau}, \gamma) & \stackrel{\text{def}}{=} \gamma, \text{ where } \gamma \text{ has no tense operators} \\
\mathsf{TransMonitor}(\overline{\tau}, \phi \, \mathsf{U}_{[\tau,\tau']} \, \psi) & \stackrel{\text{def}}{=} \exists t[\overline{\tau} + \tau \le t \le \overline{\tau} + \tau' \wedge \\
 & \quad \mathsf{TransMonitor}(t, \psi) \wedge \forall t'[\overline{\tau} \le t' < t \rightarrow \mathsf{TransMonitor}(t', \phi)]] \\
\mathsf{TransMonitor}(\overline{\tau}, \Box_{[\tau,\tau']} \, \phi) & \stackrel{\text{def}}{=} \forall t[\overline{\tau} + \tau \le t \le \overline{\tau} + \tau' \rightarrow \mathsf{TransMonitor}(t, \phi)] \\
\mathsf{TransMonitor}(\overline{\tau}, \Diamond_{[\tau,\tau']} \, \phi) & \stackrel{\text{def}}{=} \exists t[\overline{\tau} + \tau \le t \le \overline{\tau} + \tau' \wedge \mathsf{TransMonitor}(t, \phi)]
\end{array}$$

Line 4 ($f \hat{=} v$) handles elementary fluent formulas, which are not full logic formulas in themselves and require the addition of an explicit temporal context $[\overline{\tau}]$. Other formulas without occurrences of tense operators, for example value comparisons of the form $v = w$, require no temporal context and are handled in line 5.

The *Trans* translation function is extended for tense monitor formulas by defining $\mathit{Trans}(\gamma) = \mathit{Trans}(\mathsf{TransMonitor}(0, \gamma))$.

A few examples may be in order.

Example 3 Suppose that whenever a UAV is moving, the winch should not be lowered. In this example, "moving" should not be interpreted as "having a speed not identical to zero", since the UAV might not be able to stay perfectly still when hovering and sensor noise may cause the sensed speed to fluctuate slightly. Instead, the UAV is considered to be still when its sensed speed is at most s_{min}. We assume the existence of a winch feature representing how far the winch has been extended and a limit w_{min} determining when the winch is considered to be lowered, which leads to the following monitor formula.

$$\Box\, \forall uav.\mathsf{speed}(uav) > s_{\mathsf{min}} \rightarrow \mathsf{winch}(uav) \leq w_{\mathsf{min}} \qquad \Box$$

Note that this does not in itself *cause* the UAV to behave in the desired manner. This has to be achieved in the lower level implementations of the helicopter control software. This monitor formula instead serves as a method for detecting the failure of the helicopter control software to function according to specifications.

Example 4 Suppose that a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification:

$$\begin{aligned}&\Box\, \forall uav.(\mathsf{power}(uav) > M \rightarrow \\ &\qquad \mathsf{power} < f \cdot M \underset{[0,\tau]}{\mathsf{U}} \underset{[0,\tau']}{\Box} \mathsf{power}(uav) \leq M) \qquad \Box\end{aligned}$$

Further examples will be shown in Sect. 9, after the introduction of operator-specific monitor formulas and a means for formulas to explicitly represent queries about aspects of the execution state of the autonomous system.

8.3 Checking monitor conditions using formula progression

We now have a syntax and a semantics for conditions to be monitored during execution. Given the complete state sequence corresponding to the events taking place during the execution of a plan, a straight-forward implementation of the semantics can be used to test whether a monitor formula is violated. This is sufficient for post-execution analysis, but true execution monitoring requires prompt detection of potential or actual failures *during* execution.

A formula progression algorithm can be used for this purpose [4,5]. By definition, a formula ϕ holds in the state sequence $[s_0, s_1, \ldots, s_n]$ iff $\mathsf{Progress}(\phi, s_0)$ holds in $[s_1, \ldots, s_n]$. Thus, a monitor formula can be incrementally progressed through each new state that arrives from DyKnow, evaluating only those parts of the formula that refer to the newly received state.

As soon as sufficient information has been received to determine that the monitor formula must be violated regardless of the future development of the world, the formula $\bot$ (false) is returned. For example, this will happen as soon as the formula $\Box\, \mathsf{speed} < 50$ is progressed through a state where $\mathsf{speed} \geq 50$. Using progression thus ensures that failures are detected quickly and without evaluating formulas in the same state more than once.

The result of progression might also be $\top$ (true), in which case the formula must hold regardless of what happens "in the future". This will occur if the formula is of the form $\Diamond\, \phi$ (eventually, ϕ will hold), and one has reached a state where ϕ indeed does hold. In other cases, the state sequence will comply with the constraint "so far", and progression will return a new and potentially modified formula that should be progressed again as soon as another state is available.

Since states are not first-class objects in TAL, the state-based definition of progression must be altered slightly. Instead of taking a state as an argument, the procedure below is provided with an interpretation together with the timepoint corresponding to the state through which the formula should be progressed.

An additional change improves performance and flexibility for the situation where a single sample period corresponds to multiple discrete TAL timepoints. Specifically, if samples are known to arrive every m timepoints and one is progressing a formula ϕ where the lower temporal bounds of all tense operators are multiples of m', it is possible to progress ϕ through $n = \gcd(m, m')$ timepoints in a single step. The value of n is assumed to be provided to the progression procedure as defined below. This permits the temporal unit to be completely decoupled from the sample rate while at the same time retaining the standard TAL-based semantics, where states exist at every discrete timepoint.

For example, suppose a TAL timepoint corresponds to 1 ms. If samples arrive every 50 ms, the formula $\Diamond_{[0,3037]}\ \phi$ can be progressed through $\gcd(50, 0) = 50$ timepoints in a single step, while the formula $\Diamond_{[40,3037]}\ \phi$ can be progressed through $\gcd(50, 40) = 10$ timepoints. If samples arrive every 100 ms, the formula $\Diamond_{[0,3037]}\ \phi$ can be progressed through $\gcd(100, 0) = 100$ timepoints in a single step, while the formula $\Diamond_{[40,3037]}\ \phi$ can be progressed through $\gcd(100, 40) = 20$ timepoints. Thus, formula definitions and sample rates can be altered independently (which would not be the case if a TAL timepoint was defined to be a single sample interval), and progression automatically adapts to the current situation.

The progression algorithm below satisfies the following property. Let n be a progression step measured in timepoints, ϕ be a monitor formula where all times are multiples of n, τ a numeric timepoint, and $\mathcal{I}$ a TAL interpretation. Then, $\mathsf{Progress}(\phi, \tau, n, \mathcal{I})$ will hold at $\tau + n$ in $\mathcal{I}$ iff ϕ holds at τ in $\mathcal{I}$. More formally,

$$\mathcal{I} \models \mathit{Trans}(\mathsf{TransMonitor}(\tau, \phi)) \text{ iff } \mathcal{I} \models \mathit{Trans}(\mathsf{TransMonitor}(\tau + n, \mathsf{Progress}\\ (\phi, \tau, n, \mathcal{I}))),$$

where $\mathsf{TransMonitor}(\tau, \phi)$ is the translation of a monitor formula ϕ into $\mathcal{L}(\mathrm{FL})$ relative to the timepoint τ as described above.

Definition 4 (*Progression of monitor formulas*) The following algorithm is used for progression of monitor formulas. Special cases for $\Box$ and $\Diamond$ can also be introduced for performance.

```
1  procedure Progress($\phi, \tau, n, \mathcal{I}$)
2  if $\phi = f(\overline{x}) \hat{=} v$
3    if $\mathcal{I} \models \mathit{Trans}([\tau]\ \phi)$ return $\top$ else return $\bot$
4  if $\phi = \neg\phi_1$ return $\neg$Progress($\phi_1, \tau, n, \mathcal{I}$)
5  if $\phi = \phi_1 \otimes \phi_2$ return Progress($\phi_1, \tau, n, \mathcal{I}$) $\otimes$ Progress($\phi_2, \tau, n, \mathcal{I}$)
6  if $\phi = \forall x.\phi$ // where $x$ belongs to the finite domain $X$
7    return $\bigwedge_{c \in X}$ Progress($\phi[x \mapsto c], \tau, n, \mathcal{I}$)
8  if $\phi = \exists x.\phi$ // where $x$ belongs to the finite domain $X$
9    return $\bigvee_{c \in X}$ Progress($\phi[x \mapsto c], \tau, n, \mathcal{I}$)
10 if $\phi$ contains no tense operator
11   if $\mathcal{I} \models \mathit{Trans}(\phi)$ return $\top$ else return $\bot$
12 if $\phi = \phi_1\ \mathsf{U}_{[\tau_1,\tau_2]}\ \phi_2$
13   if $\tau_2 < 0$ return $\bot$
14   elsif $0 \in [\tau_1, \tau_2]$ return Progress($\phi_2, \tau, n, \mathcal{I}$) $\vee$
15        (Progress($\phi_1, \tau, n, \mathcal{I}$) $\wedge$ ($\phi_1\ \mathsf{U}_{[\tau_1-n,\tau_2-n]}\ \phi_2$))
16   else return Progress($\phi_1, \tau, n, \mathcal{I}$) $\wedge$ ($\phi_1\ \mathsf{U}_{[\tau_1-n,\tau_2-n]}\ \phi_2$)
```

The result of Progressis simplified using the rules $\neg\bot = \top$, $(\bot \wedge \alpha) = (\alpha \wedge \bot) = \bot$, $(\bot \vee \alpha) = (\alpha \vee \bot) = \alpha$, $\neg\top = \bot$, $(\top \wedge \alpha) = (\alpha \wedge \top) = \alpha$, and $(\top \vee \alpha) = (\alpha \vee \top) = \top$. Further simplification is possible using identities such as $\Diamond_{[0,\tau]}\ \phi \wedge \Diamond_{[0,\tau']}\ \phi \equiv \Diamond_{[0,\min(\tau,\tau'')]}\ \phi$.

8.4 Recovery from failures

Any monitor formula violation signals a potential or actual failure from which the system must attempt to recover in order to achieve its designated goals.

Recovery is a complex topic, especially when combined with the stringent safety regulations associated with flying an autonomous unmanned vehicle and the possibility of having time-dependent goals as well as time-dependent constraints on the behavior of the vehicle. For example, a UAV might only be allowed to fly in certain areas at certain times. There may also be complex temporal dependencies between operations intended to be carried out by different UAVs, and given that one UAV has failed, optimal or near-optimal behavior for the aggregated system might require further modifications to the plan of another UAV. For example, if **heli1** fails to deliver a box of medicine on time, **heli2** might have to be rerouted in order to meet a goal deadline. For these reasons, our first iteration of the recovery system has not tackled incremental plan repair and modifications, even though such properties may be desirable in the long run. Instead, recovery is mainly performed through replanning, where a single planning domain specification and planning algorithm can be used for both the initial planning phase and the recovery phase. Given that the planner is sufficiently fast when generating new plans, this does not adversely affect the execution of a fully autonomous mission.

Thus, having detected a failure, the first action of a UAV is to cancel the current plan, execute an emergency break if required, and then go into autonomous hover mode. Currently, we take advantage of the fact that our UAV is rotor-based and can hover. For fixed-wing platforms, this is not an option and one would have to go into a loiter mode if the recovery involves time-consuming computation.

This is followed by the execution of a *recovery operator*, if one is associated with the violated monitor formula. The recovery operator can serve two purposes: It can specify emergency recovery procedures that must be initiated immediately without waiting for replanning, and it can permit the execution system to adjust its assumptions about what can and cannot be done. For example, if a UAV fails to take off with a certain carrier, it may have to adjust its assumptions about how many boxes it is able to lift (or, equivalently, how heavy the boxes on the carrier are). The associated recovery operator can perform this adjustment, feeding back information from the failure into the information given to the planner for replanning. The implementation of a recovery operator can also detect the fact that the UAV has attempted and failed to recover from the same fault too many times and choose whether to give up, try another method, remove some goals in order to succeed with the remaining goals, or contact a human for further guidance.

After executing a recovery operator, the UAV must find sufficient information about its environment to construct a new initial state to be given to the planner. In the simulated execution system, this information can be extracted by DyKnow directly from the simulator; even though a box may have bounced and rolled after having been dropped, the simulator will obviously know exactly where it ended up. In the real world, locating and identifying objects is a more complex topic which has not yet been tested in the context of execution monitoring and replanning. Possibly, the next iteration of the recovery system will include some degree of intervention by a human operator in order to locate objects, or confirm a UAV's hypotheses regarding object locations, before we move on to completely autonomous recovery. In either case, the topic of object identification is beyond the scope of this article.

Having constructed a formal representation of the current state, the plan executor can once again call the planner and replan from this state. This yields a new plan, which must take into account the new situation as well as any time-related constraints.

9 Further integration of planning and monitoring

Making full use of the execution monitoring system developed in the previous section requires a higher degree of integration between the planning phase and the execution monitoring phase. The following example illustrates some of the difficulties associated with doing execution monitoring without such integration.

Example 5 Whenever a UAV attaches to a carryable, it should remain attached until explicitly detached. If only global monitor formulas are available, this condition must be approximated using knowledge about the behavior of the UAV when it is in flight and when it attaches or detaches a carryable. Suppose that UAVs always descend below 4 m to attach a carryable, always stay (considerably) above an altitude of 7 m while in flight, and always descend to below 4 m before detaching a carryable. The following condition states that whenever a UAV is attached to a carryable and has achieved the proper flight altitude, it must remain attached to the carryable until it is at the proper altitude for detaching it:

$$\begin{aligned}&\Box\, \forall uav, carryable.\\ &\quad \textsf{attached}(uav, carryable) \wedge \textsf{altitude}(uav) \geq 7.0 \rightarrow\\ &\quad \textsf{attached}(uav, carryable)\ \textsf{U}\ \textsf{altitude}(uav) \leq 4.0\end{aligned}$$

This formula is inefficient, however: It will be tested in all states and for all UAVs and boxes, regardless of whether an attach action has been performed. It is also brittle: If the UAV drops the carrier before reaching flight altitude, no failure will be signaled, because the formula only triggers once the UAV rises above an altitude of 7 m. If the margin between the two altitudes is decreased, there will be a smaller interval in which a carryable might be dropped without detection, but only at the cost of increasing the risk that the formula is triggered during a proper detach action. As soon as the UAV descends to below 4 m for any reason, the monitor will cease testing whether the carryable remains attached, even if the descent was temporary and not intended to lead to a detach action. □

In the remainder of this section, we will discuss how monitor formulas can be associated with individual operators and provided to the execution system as part of the plan, improving the flexibility and modularity of the system as well as the expressive power of the formulas (Sect. 9.1). We will also discuss how to specify monitoring formulas of different longevity, either terminating at the end of an action or continuing to monitor conditions across an arbitrary interval of time (Sect. 9.2).

9.1 Operator-specific monitor formulas

The first step in a deeper integration between planning and execution monitoring involves allowing execution monitor formulas to be explicitly associated with specific operator types. Unlike global monitor formulas, such formulas are not activated before plan execution but before the execution of a particular *step* in the plan, which provides the ability to contextualize a monitor condition relative to a particular action. An operator-specific monitor formula can also directly refer to the arguments of the associated operator. As for global formulas, a recovery action can be associated with each formula.

We are not yet ready to provide an improved formula for the motivational example above. However, to illustrate the principle, consider the following example.

Example 6 When a UAV attempts to attach to a box, the attempt may fail; therefore, the success of the action should be monitored. The attach-box operator takes four arguments: A *uav*, a *box*, and the *x* and *y* coordinates of the box. Making use of the first two arguments, the following operator-specific monitor formula may be suitable:

$$\Diamond_{[0,5000]} \Box_{[0,1000]} \mathsf{attached}(uav, box)$$

Within 5000 ms, the box should be attached to the UAV, and it should remain attached for at least 1000 ms. The latter condition is intended to protect against problems during the attachment phase, where the electromagnet might attach to the box during a very short period of time even though the ultimate result is failure. □

When a plan is generated by TALplanner, each action in the plan is annotated with a set of instantiated operator-specific monitor formulas. Continuing the previous example, the action [120000, 125000] attach-box(**heli1**, **bx7**, **127.52**, **5821.23**) would be annotated with the instantiated formula $\Diamond_{[0,5000]} \Box_{[0,1000]} \mathsf{attached}(\mathbf{heli1}, \mathbf{bx7})$. During execution, this instantiated formula is added to the execution monitor immediately before beginning execution of the attach-box action.

9.2 Execution flags

The power of monitor formulas can be extended further by also giving access to certain information about the plan execution state, in addition to the world state. In the motivational example above, one would like to state that once a carrier has been attached to the UAV, it should remain attached until the UAV intentionally detaches it, that is, *until the corresponding detach action is executed*. One may also want to state that a certain fact should *hold during* the execution of an action, or that an effect should be *achieved during* the execution of an action.

In order to allow monitor formulas to query the execution state of the agent, we introduce the use of *execution flags*. An execution flag is a standard parameterized boolean fluent which holds exactly when the corresponding operator is being executed with a specific set of arguments. By convention, this fluent will generally be named by prepending "executing-" to the name of the corresponding operator. For example, the attach-box action is associated with the executing-attach-box execution flag, which takes a subset of the operator's parameters. The execution subsystem is responsible for setting this flag when execution starts and clearing it when execution ends.

Example 7 Consider the climb-for-flying-empty(*uav*) operator, which should cause the UAV to ascend to its designated flight altitude. Here, one may wish to monitor the fact that the UAV truly ends up at its flight altitude. This can be achieved using the formula $\mathsf{executing\text{-}climb\text{-}for\text{-}flying\text{-}empty}(uav)\ \mathsf{U}\ \mathsf{altitude}(uav) \geq 7.0$. □

When the operator is clear from context, we will often use the shorthand notation EXEC to refer to its associated execution flag fluent with default parameters. Using this notation, the formula above is written as $\textsc{exec}\ \mathsf{U}\ \mathsf{altitude}(uav) \geq 7.0$.

Example 8 (continued) Whenever a UAV attaches to a box, it should become attached within 5000 ms and remain attached until explicitly detached. Using execution flags in an operator-specific monitor formula for the attach-box action, this can be expressed as follows:

$$\textsc{exec}\ \mathsf{U}_{[0,5000]} (\mathsf{attached}(uav, box)\ \mathsf{U}\ \mathsf{executing\text{-}detach\text{-}box}(uav, box))$$

Compared to the motivational example, this formula is more efficient, since it is only tested when an attach action has been performed and only for the relevant UAV and box. The formula is also more robust, since failures will be signaled even if the box is dropped before flight altitude and regardless of the flight altitude of the UAV. □

10 Automatic generation of monitor formulas

The use of a single logical formalism for modeling both planning and execution monitoring provides ample opportunities for the automatic generation of conditions to be monitored. Not only do actions have preconditions that must hold and effects that must take place, but it is also possible to analyze a complete plan and generate a set of links between actions, links where the effects of one action must persist over time until used as the precondition of one or more later actions. The ability to extract these conditions from a planning domain and transfer them to an execution monitor operating within the same formalism eliminates many potential sources of inconsistencies and inaccuracies.

Preconditions: Any operator is associated with a precondition formula ϕ. Given this formula, the operator-specific monitor condition ϕ can be generated: The precondition must hold immediately when the operator is invoked.

Prevail conditions: An operator can also be associated with a "prevail condition" formula stating a condition ϕ that must hold throughout the execution of the action, as opposed to only holding in the first state. Then, the operator-specific condition $(\text{EXEC} \wedge \phi) \mathsf{U} \neg\text{EXEC}$ can be used.

Effects: The condition that the effect ϕ is achieved at some time during the execution of an action can be expressed using the monitor formula $\text{EXEC}\,\mathsf{U}\,\phi$. This, however, does not ensure that the effect still holds at the *end* of the action: It can be achieved at an intermediate point and then destroyed. The more elaborate formula $\text{EXEC}\,\mathsf{U}(\neg\text{EXEC} \wedge \phi)$ can be used to ensure that ϕ holds after the transition from execution to not executing. The formula $\text{EXEC}\,\mathsf{U}$ $(\phi\,\mathsf{U}\,\neg\text{EXEC})$ can be used to express the same condition: The operator must execute until ϕ holds, after which ϕ must hold until the operator is no longer executing.

Temporal constraints: An operator can be associated with constraints on the duration of its execution. Such constraints can be viewed in two different ways, both of which are supported by TALplanner: As a specification of the *most likely* behavior of the operator, which can be used to provide a reasonable estimate of the time required to execute a plan, or as a *definite* constraint on how much time can possibly be required if the operator is working as intended. In the latter case, the constraint can be used to generate a monitor formula constraining the amount of time that can pass before $\neg\text{EXEC}$ holds.

Causal links: An effect monitor can ensure that the desired effects of an action materializes in the real world after the execution of the action. If this effect is later used as a precondition of another action, a precondition monitor can be used to ensure that the effect still holds—but between the effect and the precondition, a considerable amount of time may have passed. For example, **heli1** may use attach-box to pick up **bx3**, which gives rise to an effect monitor ensuring that attached(**heli1**, **bx3**) holds. Then, it may ascend to flight altitude, fly for several minutes towards its destination, descend, and use detach-box to put the box down. Only when detach-box is executed does the associated precondition monitor check that attached(**heli1**, **bx3**) is still true. If the UAV dropped the box during flight, it should have been possible to detect this much earlier.

This can be done by analyzing the complete plan and generating a set of causal links between different actions. In this example, such an analysis would have detected the fact that attached(**heli1**, **bx3**) is made true by attach-box, is required by detach-box, and is not altered by any action in between. Using execution flags, this global analysis can then attach the formula executing-attach-box(**heli1**, **bx3**) U(attached(**heli1**, **bx3**) U executing-detach-box(**heli1**, **bx3**)) to this specific instance of the attach-box operator in the plan.

It should be noted that this is highly dependent on having a sufficiently detailed description of the *intermediate* effects of any action: If an operator might change a value during its execution and then restores it before its end, the domain description must correctly model this in the operator description. In the example above, this would correspond to an operator which may temporarily put down **bx3** but is guaranteed to pick it up again. This should not be seen as a severe restriction: A rich and detailed domain model is also required for many other purposes, including but not limited to concurrent plan generation.

Alternatively, if not all intermediate effects are modeled but one has an upper bound τ on how long an intermediate action may legitimately "destroy" a condition ϕ, a formula such as $\text{EXEC1}\ \mathsf{U}(\phi \wedge ((\neg\text{EXEC2} \wedge \Diamond_{[0,\tau]}\,\phi)\ \mathsf{U}(\text{EXEC2} \wedge \phi)))$ can be used: Operator 1 executes until it reaches a state where (a) the desired condition holds, meaning that the effect did take place, and (b) there is an interval of time where the operator 2 is not executing and where ϕ is always restored within τ units of time, followed by a state where operator 2 does execute and ϕ does hold.

10.1 Pragmatic generation of monitor formulas

When one works on the automatic generation of monitor conditions, there is a strong temptation to generate every condition one can possibly find, without exception. After all, it would seem that the stronger the conditions one can place on execution, the better, and exceptions make algorithms more complex and less elegant. But in a system intended to operate in the real world, pragmatism is paramount, and one must take into account several reasons why some conditions should *not* be monitored: Some violations are not fatal, some information about the environment may be expensive or difficult to sense, and, sensing may require special actions that interfere with normal mission operations. Additionally, the introduction of a richer and more detailed domain model should not automatically lead to heavier use of sensors.

For these reasons, our system is mainly built on the *selective* generation of conditions to be monitored: Each precondition, prevail condition, effect, and temporal constraint can be annotated with a flag stating whether it should be monitored. This provides most of the benefits of automatic formula generation while keeping the control in the hands of the domain designer.

11 Execution monitoring with inaccurate sensors

The purpose of execution monitoring is the detection of such failures that would prevent a system from achieving its designated goals, or that would cause other types of undesirable or potentially dangerous behavior. In this process, one should not only work to maximize the probability that a failure is detected but also attempt to minimize the probability of false positives. In other words, a system should not signal a failure if none has occurred. In some cases this may be even more important than detecting non-catastrophic failures, because while such failures can prevent a system from achieving all its subgoals, a persistent false positive could cause the system to stall entirely, believing it is continuously failing.

There are several different reasons for false positives, and different approaches may be suitable for dealing with these problems. There is of course always the possibility of *catastrophic sensor failure*, where a sensor begins returning nonsensical information. Though this can perhaps be modeled and detected, we consider it outside the scope of the article. Instead, we focus on the difficulties that are present even when sensors are functioning nominally. Our architecture being a distributed system, there is always a possibility of *dropouts* and *delays* where sensor data temporarily disappears due to a communication failure. Similarly, events may propagate through different paths and arrive *out of order*. Also, one must always expect a certain amount of *noise* in sensor values.

These difficulties can be ameliorated through careful state generation. Temporary dropouts can be handled through extrapolating historical values. Delays can be handled by waiting until time $n + m$ before progressing a formula through the state at time n; sensor values from time n then have an additional m milliseconds to propagate before the state generator assumes that all values must have arrived. This obviously has the cost of also delaying failure detection by m milliseconds, which requires a careful consideration of the compromise to be made between prompt detection and the probability of false positives. Noise could conceivably be minimized through sensor value smoothing techniques and sensor fusion techniques where the measurements from several sensors are taken into account to provide the best possible estimation.

Though additional measures could be considered, it follows from the fundamental nature of a distributed system with noisy sensors that the possibility of inaccuracies in the detected state sequence can never be completely eliminated. Instead, we suggest a two-fold approach to minimizing false positives: Be careful when generating states, but also be aware that state values may be inaccurate and take this into consideration when writing monitor formulas.

Consider, as an example, the condition $\Box\, \forall uav.\mathsf{speed}(uav) \leq T$. On the surface, the meaning of this formula would seem to be that the speed of a UAV must never exceed the threshold T. However, this formula will not be evaluated in the real world: It will be evaluated in the state sequence approximation generated by DyKnow, and there, its meaning will be that the *sensed and approximated* speed of a UAV must never exceed the threshold T. Since a single observation of $\mathsf{speed}(uav)$ above the threshold might be an error or a temporary artifact, a more robust solution would be to signal a failure if the sensed speed has been above the threshold during an interval $[0, \tau]$ instead of at a single timepoint. This can be expressed as $\Box \Diamond_{[0,\tau]} \mathsf{speed}(uav) \leq T$: It should always be the case that within the interval $[0, \tau]$ from now, the sensed speed returns to being below the threshold.

Since the formula above only requires that a single measurement in every interval of length τ must be below the threshold, it might be considered too weak for some purposes. An alternative would be to require that within τ time units, there will be an *interval* of length τ' during which the UAV stays within the limits: $\Box\, \mathsf{speed}(uav) < T \rightarrow \Diamond_{[0,\tau]} \Box_{[0,\tau']} \mathsf{speed}(uav) \leq T$.

12 Empirical evaluation of the formula progressor

For our approach to be useful, it must be possible to progress typical monitor formulas through the state sequences generated during a mission using the often limited computational power available in an autonomous robotic system, such as the DRC computer on board our mobile UAV platforms. Early testing in our emergency services planning domain, both in flight tests and in hardware-in-the-loop simulation, indicated that the progression system had more than sufficient performance for the formulas we used, requiring only a fraction of the available processing power. If there is a bottleneck, then it lies not in the use of formula progression but in actually retrieving and processing the necessary sensory data, which is necessary regardless of the specific approach being used for monitoring execution and is therefore outside the purview of this paper.

Nevertheless, the processing requirements for the progression of typical monitor formulas should be quantified in some manner. Our evaluations build on experiments with the emergency services planning domain as well as the use of synthetic tests. For the remainder of this section, we will focus on the synthetic tests, where we can study highly complex combinations of time and modality. We use the actual DRC computer on board an RMAX UAV to run progression tests for formulas having forms that are typical for monitor formulas in our application. State sequences are constructed to exercise both the best and the worst cases for these formulas.

Results are reported both in terms of the average time required to progress a certain number of formulas through each state in a sequence and in terms of how progression time changes across states in the sequence, due to formula expansion during progression. This can be directly translated into the number of formulas that can be progressed on the DRC computer given a specific state sampling rate. The final objective, of course, is not to progress a vast number of formulas but to ensure that the required formulas can be progressed using only a small percentage of the CPU, leaving enough time for other processes and services to run.

Depending on the domain and the execution properties that should be modeled, the number of formulas that need to be progressed concurrently will vary. In our work with the emergency services application, we observed the use of from one or two up to at most a few dozen formulas at any given time in the planning context, with the average being towards the low end of the scale. Since the execution monitor is a system service, other services may also have monitoring requests, so many more formulas may need to be progressed concurrently.

In all experiments, states were generated with a sampling period of 100 ms.

12.1 Experiment: always eventually

In the first experiment, we used a formula of the common form $\Box \Diamond_{[0,1000]} p$, where p is a single predicate, corresponding to the fact that p must never remain false for more than one second. This was progressed through several different state sequences, carefully constructed to exercise various progression behaviors. In the following, let $\phi = \Box \Diamond_{[0,1000]} p$.

- **(true)** – p is always true. This is the best case in terms of performance, since each time $\Diamond_{[0,1000]} p$ is progressed through a state, it immediately "collapses" into $\top$. What remains to evaluate in the next state is the original formula ϕ.
- **(true,false)** – p alternates between being true and being false. This is an intermediate case, where every "false" state results in the formula $\Diamond_{[0,1000]} p$ being conjoined to the current formula, and where this subformula collapses into $\top$ in every "true" state.
- **(false*10,true)** – p remains false for 10 consecutive sampling periods (1000 ms), after which it is true in a single sample. The sequence then repeats. Had p remained false for 11 consecutive samples, the formula would have been progressed to $\bot$, a violation would have been signaled and progression would have stopped. Consequently, this corresponds to the worst case.

The results shown in Fig. 14 can be expressed in a number of ways. If the sample period is 100 ms, all formulas must be progressed within 100 ms or the progressor will "fall behind" as new states arrive. In this situation, approximately 2500 formulas can be used even with the worst case state sequence. From another perspective, if there will be a maximum of 10

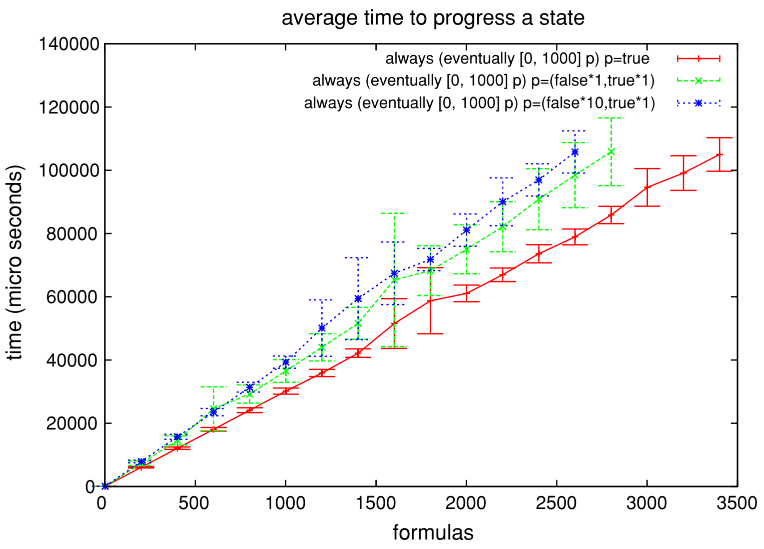


Fig. 14 Testing: always eventually (average progression time)

formulas of this type to progress at any given time, then this will require up to 0.4% of the CPU time on the DRC computer.

Results will also vary depending on the complexity of the inner formula inside the tense operators. Though the figure refers to tests run using a single fluent p, even the most complex inner formulas used in the UAV logistics domain require only a small constant multiple of the time shown in this graph.

It should be noted that the time required to progress a formula through a state is not always constant over time. The formulas themselves may vary as they are progressed through different states, and time requirements for progression vary accordingly. The average progression time must be sufficiently low, or the progressor will fall behind permanently. The *maximum* progression time should also be sufficiently low, or the progressor will *temporarily* fall behind. Figure 15 shows the precise time requirements for progressing 1000 instances of the formula $\Box \Diamond_{[0,1000]} p$ through each numbered state sample, using two different state sequences.

- In the best case, where p is always true, progression time is constant at around 30 µs per formula for a state sequence where p is always true. This is exactly what would be expected: The result of progressing the formula $\Box \Diamond_{[0,1000]} p$ through a state where p is true should always be the formula itself.
- In the worst case, where p is false for 10 sample periods and then true, $\Box \Diamond_{[0,1000]} p$ is initially progressed to the somewhat more complex formula $\Diamond_{[-100,900]} p \wedge \Box \Diamond_{[0,1000]} p$, indicating that p must become true within 900 ms. In the next state, p remains false, and the formula is further progressed to $\Diamond_{[-100,900]} p \wedge \Diamond_{[-200,800]} p \wedge \Box \Diamond_{[0,1000]} p$. This formula states that p must become true within 900 ms *and* within 800 ms, which can be reduced to the statement that it must become true within 800 ms: $\Diamond_{[-200,800]} p \wedge \Box \Diamond_{[0,1000]} p$. This creates a plateau of slightly higher progression time, which lasts until a state is reached where p is true. Then, the formula collapses back to its original form, at a slightly higher cost which results in a temporary peak in the diagram.

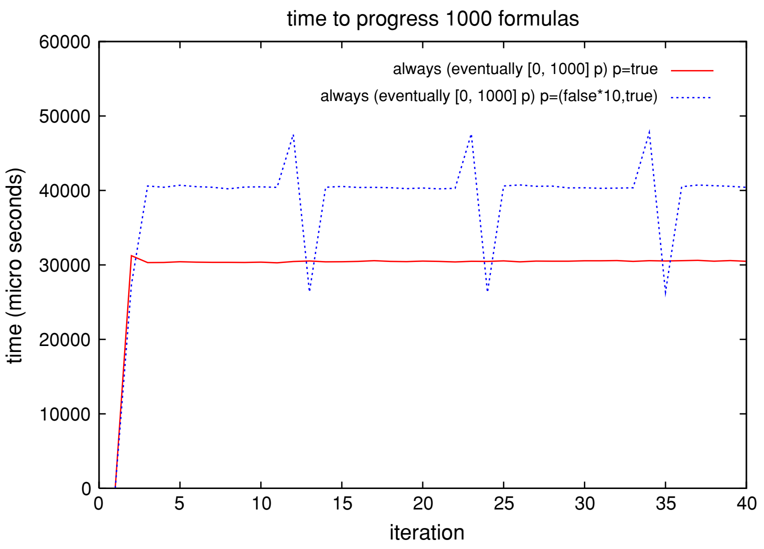


Fig. 15 Testing: always eventually (development over time)

It should be clear from Fig. 15 that these performance results are not substantially affected by using other temporal intervals than 1000 ms. Formulas of the given type do not expand arbitrarily, and changing the sequence of false and true states provided to the progression algorithm merely changes the balance between the number of timepoints where progression takes approximately 30, 40 and 47 μs. In other words, a timeout of one hour ($\Box \Diamond_{[0,3600000]} p$) is as efficiently handled as a timeout of one second.

12.2 Experiment: always not p implies eventually always

Let ϕ denote the formula $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} p$, corresponding to the fact that if p is false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true. In the second experiment, we progressed this formula through several different state sequences:

- **(true)** as the inner formula only needs to be progressed when p is false, this is the best state sequence in term of performance.
- **(false*1,true*10)** here p is false during one sample and true during ten samples, after which the sequence repeats.

  As p is initially false, the formula is initially progressed to $(\Diamond_{[0,900]} \Box_{[0,999]} p) \wedge \phi$, where the first conjunct reflects the fact that 100 ms have already passed, leaving at most 900 ms until the required interval of length 1000 where p is true.

  Progressing through the next state sample, where p is true, results in $((\Box_{[0,899]} p) \vee \Diamond_{[0,800]} \Box_{[0,999]} p) \wedge \phi$. The first conjunct of the previous formula was $\Diamond_{[0,900]} \Box_{[0,999]} p$, and as p is now true, this can be satisfied in two ways: Either a sufficiently long interval where p is true begins now, extending 900 ms from the next state ($\Box_{[0,899]} p$), or it begins later, within 800 ms ($\Diamond_{[0,800]} \Box_{[0,999]} p$).

  After simplification, any further progression through a state where p is true will still result in a formula having the form $\Diamond_{[0,i]} \Box_{[0,999]} p) \vee (\Box_{[0,j]} p)) \wedge \phi$, up to the point where p has been true for a sufficient period of time and the formula once again collapses to ϕ.

- **(false*1,true*40)** here p is false during one sample and true during forty samples, after which the sequence repeats.
- **(false*10,true*10)** here p is false during ten samples and true during ten samples, after which the sequence repeats.

The results shown in Fig. 16 shows that 100 ms is sufficient for the progression of between 1500 and 3000 formulas of this form, depending on the state sequence.

Figure 17 shows the amount of time required to process 1000 instances of the formula $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]}\, p$ through each individual state sample.

- In the case where p is always true, progression time is constant after the first progression step: The antecedent of the implication never holds, and the progression algorithm always returns the original formula. This is the best case, requiring approximately 32 μs for each progression step.
- Whenever p is false, the antecedent of the implication holds and the consequent $\Diamond_{[0,1000]}$ $\Box_{[0,999]}\, p$ is progressed and conjoined to the current formula. Thus, if p alternates between being false for 1 sample and true for 10 samples, the formula expands only once. This results in a higher but constant "plateau" followed by a collapse back to the original formula.
- If p is true for more than 10 samples, progression eventually returns to the lowest plateau, as can be seen in the case where p alternates between being false for 1 sample and true for 40 samples.
- Finally, consider the case where p remains false for as long as possible (10 samples), after which it is true for the minimum period required (10 samples). In the first state where p is false, the antecedent of the implication holds and its consequent must be conjoined, just like before. However, this is now followed by another state where p is false. As the implication is triggered again, the formula temporarily expands even more. Though the

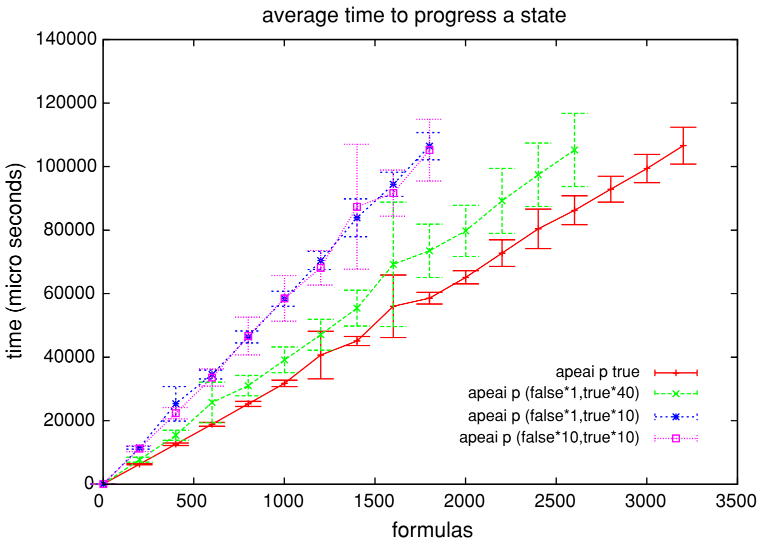


Fig. 16 Testing: always not p implies eventually always (average progression time)

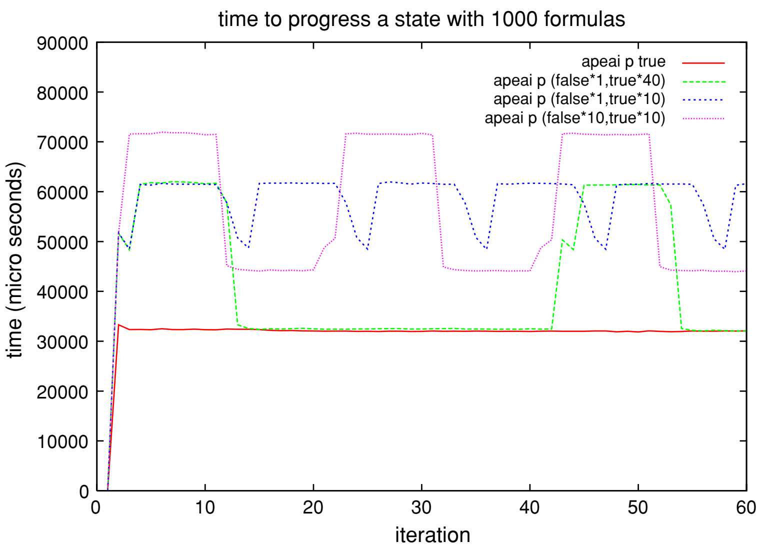


Fig. 17 Testing: always not p implies eventually always (development over time)

formula immediately collapses due to subsumption, the additional processing leads to a high plateau where a single progression step may take as much as 72 µs.

Again, the figure indicates upper and lower bounds for progression time requirements: A single progression step may take between 32 and 72 µs, and altering permitted interval lengths only changes the range of permitted proportions between these timepoints—in other words, the possible lengths of the plateaus. Regardless of interval lengths, the worst case cannot require more than 72 µs per iteration, which means that more than 1300 formulas can be progressed within a sample period of 100 ms.

13 Related work

Many architectures that deal with both planning and execution focus entirely on recovery from detected problems by plan repair or replanning [34,52,59,3,39]. These architectures usually assume that state variables are correctly updated and that plan operator implementations detect any possible failure in some unspecified manner, and thereby do not consider the full execution monitoring problem. A more elaborate and general approach is taken by Wilkins et al. [69] where a large set of different types of monitors are used in two different applications. However, in their approach monitors are procedurally encoded instead of using a declarative language.

In those architectures that do consider execution monitoring, it is often an intrinsic part of an execution component rather than integrated with the planner [30,65]. The most common approach uses a predictive model to determine what state a robot should be in, continuously comparing this to the current state as detected by sensors [10,67]. This is a well-studied problem in control theory, where it is often called fault detection and isolation (FDI) [36]. Using models derived from first principles it is possible to detect faulty sensors and other components. The same approach has also been taken in planning, where the plan itself leads to a prediction of a state sequence that should occur [32], as well as in path planning and robot navigation [30,35]. For example, Gat et al. [35] takes the output from a path planner and simulates the expected sensor readings the agent should receive when following the path. From these expectations, one derives for each sensor reading an interval of time within which the reading is expected to occur. While following the generated path, readings outside the expected interval cause the robot to switch to a recovery mode which attempts to handle the unintended situation.

Several significant weaknesses can be identified in this approach. The fact that one can detect a discrepancy between the current state and the predicted state does not necessarily mean that this discrepancy has a detrimental effect on the plan. Thus, one must take great care to distinguish essential deviations from unimportant ones, rendering the advantage of being able to automatically derive problems in execution from a predictive model considerably less significant. Similarly, that one can predict a fact does not necessarily mean that this fact must be monitored at all. Excessive monitoring may be unproblematic in a chemical processing plant where fixed sensors have been placed at suitable locations in advance and information gathering is essentially free, but does cause problems when monitoring costs are not negligible. Specifically, increasing the richness and fidelity of a domain model should not necessarily cause the costs for monitoring to increase.

The approach used by Fernández and Simmons [30] focuses on undesirable behavior rather than expected situations, explicitly defining a set of hierarchically organized monitors corresponding to *symptoms* that can be detected by a robot. Top level monitors cover many cases, but report problems with a large delay and provide little information about the cause of a problem. For example, a top level monitor may detect excessive action execution time, which is of little help if a problem occurs at the beginning of an action but covers any conceivable reason for delay. Leaf monitors are more specific and provide more information, but may provide less coverage. While the idea of focusing on explicitly specified undesirable behavior avoids the problems discussed above, symptoms appear to be hardcoded rather than declaratively specified. As our approach can monitor both operator execution time, operator-specific constraints and global constraints, the approach taken by [30] can be emulated in our system, including checking for special situations such as a robot getting stuck spinning around.

While these approaches do cover some important aspects of the execution monitoring problem, they still generally fail to consider issues related to multiple agents, the information gathering problem, and the problem of incomplete information about the current state. Another major weakness is that only the current state is considered. Adapting ideas from model checking [11] to be able to talk about sequences of states, Ben Lamine and Kabanza [8] expressed the desired properties of a system in a temporal logical formalism, whereas model checking generally tests such properties against a system model, their execution monitor system tests them against an actual execution trace. Similar ideas have also been developed in the model checking community. There, the problem of checking whether a single execution trace of a system satisfies a property is called path model checking [55,54,33], or runtime verification if the evaluation is done incrementally as the trace develops [66,62,27,6,7]. These approaches are equivalent to progression of a formula, but have been further extended to more expressive logics.

Though the work by Ben Lamine and Kabanza provided part of the inspiration for this article, it focuses on a reactive behavior-based architecture where the combined effects of a set of interactive behaviors is difficult to predict in advance. There, monitor formulas generally state global properties that cannot be monitored by internal behaviors, such as the fact that after three consecutive losses of communication, a given behavior must be active. A violation triggers an ad-hoc behavior that attempts to correct the problem. In comparison, our system is based on the use of planning, with an integrated domain description language. Formulas are not necessarily global, but can be operator-specific. Our approach provides a selective mechanism for extracting monitor formulas through automated plan analysis and supports recovery through replanning. DyKnow also gives us the possibility to provide attention focused state descriptions, where the execution monitor contextually subscribes only to those state variables that are required for progressing the currently active monitor formulas. Additionally, state sampling rates can be decided individually for each monitor formula. Combining this with the use of operator-specific monitor formulas that are only active when specific tasks are being performed ensures that state variables are only computed when strictly required, ensuring minimal usage of resources including sensors and computational power. This can be particularly beneficial in the case of state variables requiring image processing or other complex operations.

See Pettersson [60] for an overview of systems related to execution monitoring.

14 Conclusions and future work

In this article, we have presented an architectural framework for planning and execution monitoring where conditions to be monitored are specified as formulas in a temporal logic. A key point in this architecture is the use of the same logic formalism, TAL, for both planning and monitoring. This allows a higher degree of integration between these two important topics in several respects, including the fact that conditions to be monitored can be attached directly to specific actions in a plan and that a plan can be analyzed to automatically extract conditions to be monitored.

The framework we present has been implemented and integrated into our unmanned aerial vehicle platforms. The system has been deployed and used in actual missions with the UAVs in a smaller urban environment. Empirical testing has taken several different shapes during the course of the project. While an early version of this system was tested on board the UAV, the full logistics scenario cannot be tested until a winch and an electromagnet are available, together with suitably prepared boxes and carriers. These devices are under development and will be ready for use in the near future. Until then, only monitor formulas related to pure flight have been tested on the physical system, and ironically (or fortunately) the UAV system has proven quite stable and few opportunities for failure detection have presented themselves. Therefore, the most intensive tests have been performed through simulation.

In terms of testing their adequacy for detecting failures, monitor formulas have been tested through intentional fault injection, where one may, for example, simulate dropping a box or failing to take off. Surprisingly often, formulas have also been tested through unintentional failures. For example, when ordered to detach a box, the simulator does not simply place them at the designated coordinates; instead, it turns off the simulated electromagnet, after which the box falls from its current altitude towards the ground. This, in turn, may make the box bounce or roll away from its intended coordinates. Taken together, the simulated environment is most likely as good at testing a wide variety of failures as the physical UAV system, and certainly more efficient.

Finally, the computational adequacy of the system is also important. What matters in this area is not the analytically derived worst case temporal complexity for the most complex monitor formulas, but the actual worst case and average case performance for those formulas that are actually useful and relevant for a domain. Extensive testing has therefore been done by running sets of typical monitor formulas in parallel using constructed best-case and worst-case state sequences. Results indicate that while there is a certain cost associated with generating state sequences, the additional cost for each new formula is very low given the typical structure of monitor formulas. Thus, given that the appropriate sensor values are available in DyKnow, execution monitoring does not require significant additional resources, even if large numbers of monitor formulas are active concurrently.

An interesting topic for future research is that of autonomously determining the state that results from a failure. For example, given that a UAV detects that it dropped a box, where did that box end up? In the first stage, a mixed initiative approach may be appropriate, where the UAV signals a failure to a human operator which helps the UAV find the box, possibly aided by the camera on board the UAV. Ideally, the UAV should be able to find the box completely autonomously; for example, by flying a regular scanning pattern and using its vision subsystem, a laser scanner, or other remote sensing equipment to find likely candidates. Here, information about previously performed actions can potentially be of value in determining the most likely location of a box, as well as information from geographic information systems and other sources of data regarding the environment.

Based on a great deal of experience with our UAV systems, it is our strong belief that using logics as the basis for deliberative functionalities such as planning and monitoring the expected effects of plans in the environment simplifies the development of complex intelligent autonomous systems such as UAVs. Temporal Action Logic and its tense formula subset are highly expressive languages which are well suited for describing the UAV domain and for expressing the monitoring conditions we are interested in. Therefore we believe this approach provides a viable path towards even more sophisticated and capable autonomous UAV applications.

Acknowledgements This work is partially supported by grants from the Swedish Research Council (50405001, 50405002), the Swedish Aeronautics Research Council (NFFP4-S4203), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, and CENIIT, the Center for Industrial Information Technology.

References

1. Alur, R., Feder, T., & Henzinger, T. A. (1991). The benefits of relaxing punctuality. In *Proceedings of the 10th ACM symposium on principles of distributed computing (PODC-1991)* (pp. 139–152). ACM Press: Montréal.
2. Alur, R., & Henzinger, T. A. (1992). Back to the future: Towards a theory of timed regular languages. In *Proceedings of the 33rd IEEE symposium on foundations of computer science (FOCS-1992), IEEE* (pp. 177–186). Pittsburgh: IEEE Computer Society Press.
3. Ambros-Ingerson, J., & Steel, S. (1988). Integrating planning, execution and monitoring. In *Proceedings of the 7th national conference of artificial intelligence (AAAI-1988)* (pp. 83–88). St. Paul: AAAI Press/The MIT Press.
4. Bacchus, F., & Kabanza, F. (1996). Planning for temporally extended goals. In *Proceedings of the 13th national conference on artificial intelligence (AAAI-1996)* (pp. 1215–1222). Portland: AAAI Press/The MIT Press.
5. Bacchus, F., & Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence, 22*, 5–27.
6. Barringer, H., Goldberg, A., Havelund, K., & Sen, K. (2004). Program monitoring with LTL in EAGLE. In *Proceedings of the 18th international parallel and distributed processing symposium (IPDPS-2004)* (264+ pp).
7. Barringer, H., Rydeheard, D., & Havelund, K. (2007). *Runtime verification*. Chap rule systems for run-time monitoring: From Eagle to RuleR (pp. 111–125). Berlin/Heidelberg: Springer.
8. Ben Lamine, K., & Kabanza, F. (2002). Reasoning about robot actions: A model checking approach. In *Revised papers from the international seminar on advances in plan-based control of robotic agents* (pp. 123–139). Springer.

9. Bjäreland, M. (2001). Model-based execution monitoring. PhD thesis, Linköpings universitet, Linköping Studies in Science and Technology, Dissertation no. 688.
10. Chien, S., Knight, R., Stechert, A., Sherwood, R., & Rabideau, G. (2000). Using iterative repair to improve the responsiveness of planning and scheduling. In S. Chien, S. Kambhampati, & C. A. Knoblock (Eds.), *Proceedings of the 5th international conference on artificial intelligence planning systems (AIPS-2000)* (pp. 300–307). Breckenridge: AAAI Press.
11. Clarke, E. M., Grumberg, O., & Peled, D. A. (2000). *Model checking*. The MIT Press.
12. Coradeschi, S., & Saffiotti, A. (2003). An introduction to the anchoring problem. *Robotics and Autonomous Systems, 43*(2–3), 85–96.
13. De Giacomo, G., Reiter, R., & Soutchanski, M. (1998). Execution monitoring of high-level robot programs. In A. G. Cohn, L. K. Schubert, & S. C. Shapiro (Eds.), *Proceedings of the 6th international conference on principles of knowledge representation and reasoning (KR-1998)* (pp. 453-465). Trento: Morgan Kaufmann.
14. Doherty, P. (1994). Reasoning about action and change using occlusion. In A. G. Cohn (Ed.) *Proceedings of the 11th European conference on artificial intelligence (ECAI-1994)* (pp. 401–405). Chichester: John Wiley and Sons.
15. Doherty, P. (2004). Advanced research with autonomous unmanned aerial vehicles. In D. Dubois, C. A. Welty, & M. A. Williams (Eds.), *Proceedings on the 9th international conference on principles of knowledge representation and reasoning (KR-2004)*. Whistler: AAAI Press (Extended abstract for plenary talk).
16. Doherty, P. (2005). Knowledge representation and unmanned aerial vehicles. In A. Skowron, J. P. A. Barthès, L. C. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu, & N. Zhong (Eds.), *Proceedings of the 2005 IEEE/WIC/ACM international conference on intelligent agent technology (IAT-2005)* (pp. 9–16). Compiegne, France: IEEE Computer Society.
17. Doherty, P., Granlund, G., Kuchcinski, K., Sandewall, E., Nordberg, K., Skarman, E., & Wiklund, J. (2000). The WITAS unmanned aerial vehicle project. In W. Horn (Ed.), *Proceedings of the 14th European conference on artificial intelligence (ECAI-2000)* (pp. 747–755). Amsterdam: IOS Press.
18. Doherty, P., Gustafsson, J., Karlsson, L., & Kvarnström, J. (1998). TAL: Temporal Action Logics—language specification and tutorial. *Electronic Transactions on Artificial Intelligence, 2*(3–4), 273–306.
19. Doherty, P., Haslum, P., Heintz, F., Merz, T., Nyblom, P., Persson, T., & Wingman, B. (2004). A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th international symposium on distributed autonomous robotic systems (DARS-2004)* (pp. 221–230). Toulouse, France.
20. Doherty, P., & Kvarnström, J. (1999). TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In C. Dixon & M. Fisher (Eds.), *Proceedings of the 6th international workshop on temporal representation and reasoning (TIME-1999)* (pp. 47–54). Orlando: IEEE Computer Society Press.
21. Doherty, P., & Kvarnström, J. (2001). TALplanner: A temporal logic-based planner. *Artificial Intelligence Magazine, 22*(3), 95–102.
22. Doherty, P., & Kvarnström, J. (2008). Temporal action logics. In V. Lifschitz, F. van Harmelen, & F. Porter (Eds.), *The handbook of knowledge representation* (Chap. 18). Elsevier.
23. Doherty, P., Łukaszewicz, W., & A. Szałas. (1995). Computing circumscription revisited: Preliminary report. In *Proceedings of the 14th international joint conference on artificial intelligence (IJCAI-1995)* (Vol. 2, pp. 1502–1508). Montréal: Morgan Kaufmann.
24. Doherty, P., Łukaszewicz, W., & Szałas, A. (1997). Computing circumscription revisited: A reduction algorithm. *Journal of Automated Reasoning, 18*, 297–336.
25. Doherty, P., & Meyer, J. J. C. (2007). Towards a delegation framework for aerial robotic mission scenarios. In M. Klusch, K. V. Hindriks, M. P. Papazoglou, & L. Sterling (Eds.), *Proceedings of the 11th international workshop on cooperative information agents (CIA-2007)*, LNCS (pp. 5–26). Delft, The Netherlands: Springer.
26. Doherty, P., & Rudol, P. (2007). A UAV search and rescue scenario with human body detection and geolocalization. In *20th Australian joint conference on artificial intelligence (AI-2007)*, LNCS (pp. 1–13). Gold Coast, Queensland: Springer.
27. Drusinsky, D. (2003). Monitoring temporal rules combined with time series. In *Proceedings of the computer aided verification conference (CAV-2003)*, LNCS (Vol. 2725, pp. 114–118). Springer.
28. Duranti, S., Conte, G., Lundström, D., Rudol, P., Wzorek, M., & Doherty, P. (2007). LinkMAV, a prototype rotary wing micro aerial vehicle. In *Proceedings of the 17th IFAC symposium on automatic control in aerospace (ACA-2007)*. Toulouse, France.
29. Emerson, E. A. (1990). Temporal and modal logic. *Handbook of theoretical computer science, volume b: Formal models and semantics* (pp. 997–1072). MIT Press and Elsevier.

30. Fernández, J. L., & Simmons, R. G. (1998). Robust execution monitoring for navigation plans. In *Proceedings of the 1998 IEEE/RSJ international conference on intelligent robots and systems (IROS-1998)* (pp. 551–557). Victoria, BC, Canada.
31. Fichtner, M., Grossmann, A., & Thielscher, M. (2003). Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae, 57*(2–4), 371–392.
32. Fikes, R. (1971). Monitored execution of robot plans produced by STRIPS. In *Proceedings of the IFIP congress (IFIP-1971)* (pp. 189–194). Ljubljana, Yugoslavia.
33. Finkbeiner, B., & Sipma, H. (2004). Checking finite traces using alternating automata. *Formal Methods in System Design, 24*(2), 101–127.
34. Finzi, A., Ingrand, F., & Muscettola, N. (2004). Robot action planning and execution control. In *Proceedings of the 4th international workshop on planning and scheduling for space (IWPSS-2004)*. Darmstadt, Germany.
35. Gat, E., Slack, M. G., Miller, D. P., & Firby, R. J. (1990). Path planning and execution monitoring for a planetary rover. In *Proceedings of the 1990 IEEE international conference on robotics and automation* (pp. 20–25). Cincinnati: IEEE Computer Society Press.
36. Gertler, J. (1998). *Fault detection and diagnosis in engineering systems*. New York: Marcel Dekker.
37. Ghallab, M. (1996). On chronicles: Representation, on-line recognition and learning. In L. C. Aiello, J. Doyle, & S. Shapiro (Eds.), *Proceedings of the 5th international conference on principles of knowledge representation and reasoning (KR-1996)* (pp. 597–607). San Francisco: Morgan Kaufmann.
38. Gustafsson, J., & Kvarnström, J. (2004). Elaboration tolerance through object-orientation. *Artificial Intelligence, 153*, 239–285.
39. Haigh, K. Z., & Veloso, M. M. (1998). Planning, execution and learning in a robotic agent. In R. Simmons, M. Veloso, & S. Smith (Eds.), *Proceedings of the 4th international conference on artificial intelligence planning systems 1998 (AIPS-1998)* (pp. 120–127). Pittsburgh: AAAI Press.
40. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*(3), 231–274.
41. Heintz, F., & Doherty, P. (2004a). DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems, 15*(1), 3–13.
42. Heintz, F., & Doherty, P. (2004b). Managing dynamic object structures using hypothesis generation and validation. In *Proceedings of the AAAI workshop on anchoring symbols to sensor data*.
43. Heintz, F., & Doherty, P. (2006). DyKnow: A knowledge processing middleware framework and its relation to the JDL data fusion model. *Journal of Intelligent and Fuzzy Systems, 17*(4), 335–351.
44. Heintz, F., Rudol, P., & Doherty, P. (2007). From images to traffic Behavior—a UAV tracking and monitoring application. In *Proceedings of the 10th international conference on information fusion* (pp. 1–8). Quebec: ISIF, IEEE, AES.
45. Karlsson, L., & Gustafsson, J. (1999). Reasoning about concurrent interaction. *Journal of Logic and Computation, 9*(5), 623–650.
46. Kavraki, L. E., Švestka, P., Latombe, J., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transactions on Robotics and Automation, 12*(4), 566–580.
47. Koubarakis, M. (1994). Complexity results for first-order theories of temporal constraints. In J. Doyle, E. Sandewall, & P. Torasso (Eds.), *Proceedings of the 4th international conference on principles of knowledge representation and reasoning (KR-1994)* (pp. 379–390). San Francisco: Morgan Kaufmann.
48. Kuffner, J. J., & LaValle, S. M. (2000). RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE international conference on robotics and automation (ICRA-2000)* (pp. 995–1001). San Francisco, California, USA.
49. Kvarnström, J. (2002). Applying domain analysis techniques for domain-dependent control in TALplanner. In M. Ghallab, J. Hertzberg, & P. Traverso (Eds.), *Proceedings of the sixth international conference on artificial intelligence planning and scheduling (AIPS-2002)* (pp. 101–110). Menlo Park: AAAI Press.
50. Kvarnström, J. (2005). TALplanner and other extensions to temporal action logic. PhD thesis, Linköpings universitet, Linköping Studies in Science and Technology, Dissertation no. 937.
51. Kvarnström, J., & Doherty, P. (2000). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence, 30*, 119–169.
52. Lemai, S., & Ingrand, F. (2004). Interleaving temporal planning and execution in robotics domains. In *Proceedings of the 19th national conference of artificial intelligence (AAAI-2004)* (pp. 617–622). San Jose: AAAI Press.
53. Mantegazza, P., et al. (2000). RTAI: Real time application interface. *Linux Journal, 72*.
54. Markey, N., & Raskin, J. F. (2006). Model checking restricted sets of timed paths. *Theoretical Computer Science, 358*(2–3), 273–292.

55. Markey, N., & Schnoebelen, P. (2003). *Model checking a path. CONCUR 2003–concurrency theory* (pp. 251–265).
56. Merz, T. (2004). Building a system for autonomous aerial robotics research. In *Proceedings of the 5th IFAC symposium on intelligent autonomous vehicles (IAV-2004)*. Lisbon, Portugal: Elsevier.
57. Merz, T., Duranti, S., & Conte, G. (2004). Autonomous landing of an unmanned aerial helicopter based on vision and inertial sensing. In *Proceedings of the 9th international symposium on experimental robotics (ISER-2004)*. Singapore.
58. Merz, T., Rudol, P., & Wzorek, M. (2006). Control system framework for autonomous robots based on extended state machines. In *Proceedings of the international conference on autonomic and autonomous systems (ICAS-2006)*.
59. Myers, K. (1999). CPEF: Continuous planning and execution framework. *AI Magazine, 20*(4), 63–69.
60. Pettersson, O. (2005). Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems, 53*(2), 73–88.
61. Pettersson, P. O. (2006). Sampling-based path planning for an autonomous helicopter. Licentiate thesis, Linköpings universitet, Linköping.
62. Rosu, G., & Havelund, K. (2005). Rewriting-based techniques for runtime verification. *Automated Software Engineering, 12*(2), 151–197.
63. Rudol, P., & Doherty, P. (2008). Human body detection and geolocalization for UAV human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. In *Proceedings of the IEEE aerospace conference* (pp. 1–8).
64. Rudol, P., Wzorek, M., Conte, G., & Doherty, P. (2008). Micro unmanned aerial vehicle visual servoing for cooperative indoor exploration. In *Proceedings of the IEEE aerospace conference*.
65. Simmons, R., & Apfelbaum, D. (1998). A task description language for robot control. In *Proceedings of the 1998 IEEE/RSJ international conference on intelligent robots and systems (IROS-1998)* (pp. 1931–1937). Victoria, BC, Canada.
66. Thati, P., & Rosu, G. (2005). Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science, 113*, 145–162.
67. Washington, R., Golden, K., & Bresina, J. (2000). Plan execution, monitoring, and adaptation for planetary rovers. *Electronic Transactions on Artificial Intelligence, 5*(17).
68. Weyhrauch, R. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence, 13*(1–2), 133–170.
69. Wilkins, D., Lee, T., & Berry, P. (2003). Interactive execution monitoring of agent teams. *Journal of Artificial Intelligence Research, 18*, 217–261.
70. Wzorek, M., & Doherty, P. (2008). A framework for reconfigurable path planning for autonomous unmanned aerial vehicles. *Journal of Applied Artificial Intelligence* (Forthcoming).
71. Wzorek, M., Conte, G., Rudol, P., Merz, T., Duranti, S., & Doherty, P. (2006). From motion planning to control—a navigation framework for an autonomous unmanned aerial vehicle. In *Proceedings of the 21st Bristol international unmanned air vehicle systems (UAVS) conference*. University of Bristol Department of Aerospace Engineering.