HDRC3: A Distributed Hybrid Deliberative/Reactive Architecture for Unmanned Aircraft Systems

September 10, 2013

1 Affiliation

Patrick Doherty, Jonas Kvarnström, Mariusz Wzorek, Piotr Rudol, Fredrik Heintz and Gianpaolo Conte

patrick.doherty@liu.se
jonas.kvarnstrom@liu.se
mariusz.wzorek@liu.se
piotr.rudol@liu.se
fredrik.heintz@liu.se
gianpaolo.conte@liu.se

Department of Computer and Information Science, Linköping University

Linköping

Sweden

2 Abstract

This chapter presents a distributed architecture for unmanned aircraft systems that provides full integration of both low autonomy and high autonomy. The architecture has been instantiated and used in a rotor-based aerial vehicle, but is not limited to use in particular aircraft systems. Various generic functionalities essential to the integration of both low autonomy and high autonomy in a single system are isolated and described. The architecture has also been extended for use with multi-platform systems. The chapter covers the full spectrum of functionalities required for operation in missions requiring high autonomy. A control kernel is presented with diverse flight modes integrated with a navigation subsystem. Specific interfaces and languages are introduced which provide seamless transition between deliberative and reactive capability and reactive and control capability. Hierarchical Concurrent State Machines are introduced as a real-time mechanism for specifying and executing low-level reactive control. Task Specification Trees are introduced as both a declarative and procedural mechanism for specification of high-level tasks. Task planners and motion planners are described which are tightly integrated into the architecture. Generic middleware capability for specifying data and knowledge flow within the architecture based on a stream abstraction is also described. The use of temporal logic is prevalent and is used both as a specification language and as an integral part of an execution monitoring mechanism. Emphasis is placed on the robust integration and interaction between these diverse functionalities using a principled architectural framework. The architecture has been empirically tested in several complex missions, some of which are described in the chapter.

3 Introduction

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

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

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

This chapter describes a distributed software architecture that fully integrates functionality for both low and high autonomy. The architecture has been developed for a number of years and has been both tested and deployed on a number of rotor-based AV systems. The purpose of this chapter is not only to describe the instantiation of the architecture on a particular AV system, but also to isolate and describe various generic functionalities essential to the integration of low and high autonomy. These functionalities and their architectural support can be used on any robotic system, be it an aerial, ground or underwater system. Consequently, this architectural framework should be of interest to anyone developing autonomous systems.

Let us begin by providing a motivating scenario for the use of UASs. 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. Initially 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 searching for survivors in many isolated areas where road systems had become inaccessible and providing relief in the form of delivery of food, water and medical supplies.

Assume that for a particular geographic area, one had a shortage of trained helicopter and fixed-wing pilots. Assume also that one did have access to a fleet of autonomous unmanned helicopter systems with ground operation facilities and the ability to both scan for salient entities such as injured humans and deliver supplies and resources such as relief packages containing food, water and medical supplies. These systems could then provide essential assistance to emergency rescue services.

Participating UASs would require functionalities associated with both low and high autonomy operating in tight integration with each other. Additionally, this scenario would involve several UASs, so additional support would be required for collaboration between ground operation and AVs in addition to cooperation between the AVs themselves.

The complex real-world deployment outlined above would have two phases. In the first, one would require the use of as many AVs as were in the area to scan a specific region for injured and to generate a saliency map that could then be used by emergency services or other UASs to provide relief assistance in the form of medical and food supplies. In the second phase, given a particular saliency map, one would have to determine efficient ways to deliver the appropriate supplies to the different regions in an optimal manner. In essence, there are two high-level goals to be achieved:

1. Scan a specific region with available AVs to generate a saliency map consisting of geo-locations of injured civilians.
2. Put together a logistics plan based on the saliency map and available AVs to deliver medical and food supplies to appropriate locations within this region.

Ideally, one would like to specify a mission at this level of abstraction and have the UASs generate the appropriate plan or plans for the specific AVs involved based on their capabilities, resources, and availability.

This in itself is a highly complex endeavor based on the use of sophisticated automated distributed planning techniques for teams of AVs and human resources in a mixed-initiative context. An essential aspect of the higher-level or high-autonomy functionalities is the ability to clearly and succinctly specify and generate missions to be executed and to coordinate their execution in a distributed manner.

The execution of the tasks associated with such missions in the individual AVs is just as complex. An essential aspect of the lower-level or low-autonomy functionalities involves different forms of compilation and execution mechanisms in (soft) real-time that ultimately result in the coordination and use of real-time continuous control modes associated with the individual AVs. Most importantly, the tight integration of processes at these different levels of abstraction is fundamental to the robust operation of AVs in missions which require such high degrees of autonomy.

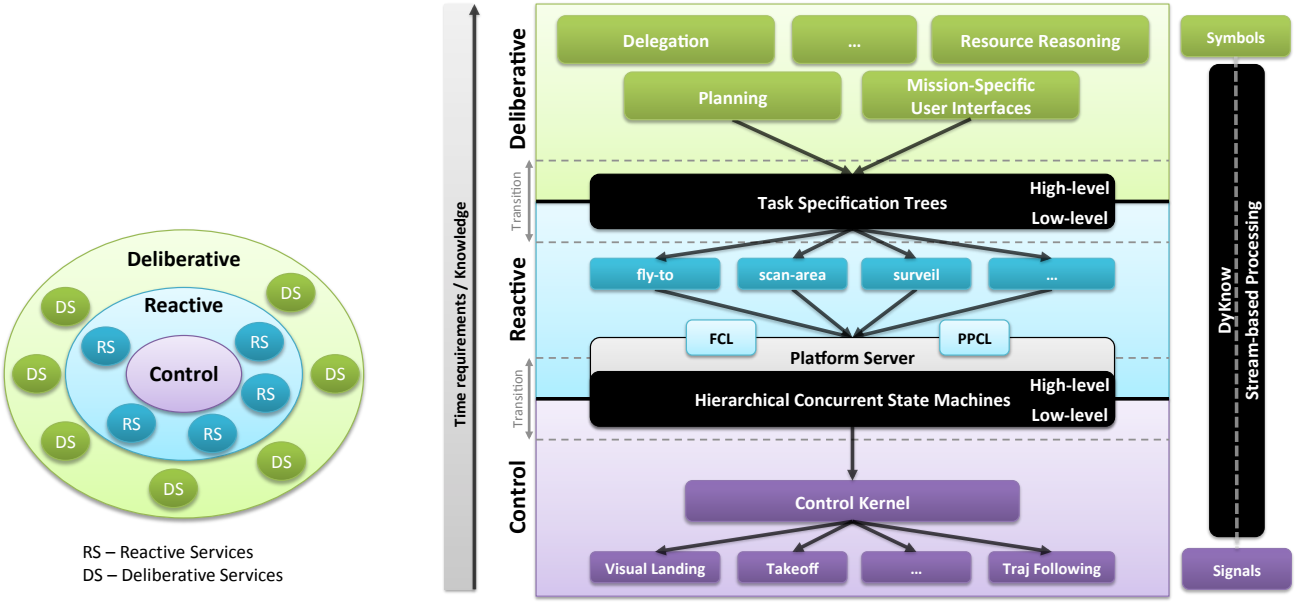
Later in the chapter, additional detail will be provided as to how the architecture supports the deployment of AVs in missions of this complexity. Before this, there will be an overview of the architecture and the functionalities it includes and a roadmap based on this overview. The remainder of the chapter will provide more detailed descriptions of these functionalities and their integration with each other.

3.1 Overview of the HDRC3 Architecture

In recent years, there has been a consolidation of thought regarding essential conceptual components required in software architectures for mobile robots, although there is wide variation in the way these components are implemented in specific robotic systems. One of the clearest presentations of this consolidation of thought is described in Gat (1997). In this article, he traces the history and development of robotic architectures in artificial intelligence and the transition from sense-plan-act architectures to what are now generically known as three-layered architectures. In his own words,

> The three-layer architecture arises from the empirical observation that effective algorithms for controlling mobile robots tend to fall into three distinct categories: (1) reactive control algorithms which map sensors directly onto actuators with little or no internal state; (2) algorithms for governing routine sequences of activity which rely extensively on internal state but perform no search; and (3) time-consuming (relative to rate of change of the environment) search-based algorithms such as planners. (Gat, 1997, p. 209)

The algorithms associated with each of these categories have distinct temporal latencies in the decision cycles: (1) millisecond range; (2) seconds range and; (3) seconds or even minutes range. Each of these computational abstractions require interfaces to each other which in essence implement compilational transitions between the layers.



(a) The concentric view

(b) The layered view

Figure 1: The structure of the Hybrid Deliberative/Reactive (HDRC3) architecture.

The Hybrid Deliberative/Reactive (HDRC3) Architecture presented here is based conceptually on the three-layered approach. It is important to state that this is due not only to the theoretical clarity of the approach but also to empirical necessity. In the development of architectures which support high and low autonomy and their integration for use in complex missions, one naturally gravitates towards such architectures. Although conceptually layered, in practice, the HDRC3 architecture is better described as concentric (as depicted in Figure 1(a)) in the sense that data and control flow are both vertical and horizontal within and across layers and that the processes evoked by the various algorithms are highly concurrent and distributed.

Figure 1(b) provides a conceptual and functional depiction of the HDRC3 architecture. It also provides a visual depiction of the roadmap for the chapter where a bottom-up approach will be used to describe the architectural components. The HDRC3 architecture consists of three functional layers and a number of interfaces between the layers:

Control Layer This layer provides a library of control modes based on the implementation of continuous control laws such as takeoff, trajectory following and vision-based landing. Algorithms associated with this layer generally have little or no internal state. Additionally, it provides a real-time environment for accessing hardware and sensors, executing control modes and switching between them.

Reactive Layer This layer coordinates the execution of high-level plans generated by the deliberative layer and implements a set of robotic behaviors such as **fly-to** and **scan-area** that may be viewed as compiled plans or as reactive procedures. These are specified as Task Specification Trees (TSTs). The reactive layer also includes a thread-based executor for TSTs. Algorithms associated with this layer do have internal state, but of a postdictive nature generally in terms of previous states of actions.

Deliberative Layer This layer provides a rich set of algorithms commonly associated with deliberation such as automated task planners, motion planners, reasoning and diagnosis mechanisms and execution monitoring mechanisms. Algorithms associated with this layer generally have rich internal state of both post- and predictive nature and often include search through large state spaces.

The interfaces used to transition between these layers are central to the success of such architectures. The HDRC3 architecture provides a rich set of interfacing mechanisms:

Interfaces between the Control Layer and the Reactive Layer A specialized language and implementation is used for interfacing between the control layer and the reactive layer. Hierarchical Concurrent State Machines are used to specify and implement mode switching behaviors at the control layer. Additionally they are used to implement low-level reactive behaviors normally associated with flight control or perception control.

Platform Server Interface Helicopter pilots and ground control personal often think of controlling unmanned aircraft in terms of a Flight Control Language (FCL) representing various flight modes and a Payload and Perception Control Language (PPCL) representing various modes of perception and sensor control. Because this abstraction is so powerful, an independent server exists which implements this abstraction using two well-defined languages for flight control and perception control, respectively. Any functionality in the system can access the server through these abstract languages.

Interfaces between the Reactive and Deliberative Layers The notion of robot task specifications and the processes they invoke are central to the achievement of goal-directed behavior. Tasks can be specified in many different ways. HCSMs are one way to specify low-level tasks. At the other end of the spectrum are highly complex tasks which use low-level tasks as primitives. Task Specification Trees (TSTs) are used as a means of not only specifying such high-level tasks declaratively, but also providing procedural correlates executable in the system. TSTs therefore provide both a declarative and procedural means of transitioning between the deliberative and reactive layers. It is often the case for instance that TSTs are used to sequentialize commands associated with the FCL and PPCL. Additionally, the output of an automated planner may be viewed as a TST.

3.1.1 Middleware Infrastructure

The functionality in the reactive and deliberative layers of the HDRC3 architecture should be viewed as sets of loosely coupled distributed processes that are highly concurrent and often require asynchronous communication with each other. These processes run on multiple on-board (or ground-based) computers, communicating through service calls and transmitting information through a set of distributed communication channels. This requires a rich middleware infrastructure to provide the communication channels between these processes and the ability of each to use other processes when required. An example would be interaction between task planners, path planners and execution monitors.

In early iterations of the HDRC3 architecture, the underlying middleware infrastructure was based on the use of CORBA (Common Object Request Broker Architecture). The architecture has recently been transitioned to using ROS, the Robot Operating System. This is a convenient open-source framework for robot software development that provides client libraries and tools for several programming languages (Quigley et al., 2009).

Software written for ROS is organized into *packages* which contain nodes, libraries and configurations. *Nodes* represent computational processes in the system and are written using the client libraries. For example, many capabilities such as task planning and motion planning are realized as separate ROS nodes. These nodes communicate by passing structured messages on *topics* which can be seen as named buses to which nodes can subscribe, and by using request/reply communication through *services*.

Although emphasis in this chapter is placed on a single platform using the HDRC3 architecture, it is also set up for collaborative missions with multiple AV systems. All agents in such a collaborative system currently use a common ROS Master, a standard ROS functionality providing registration and lookup services. For disconnected operation, a federated approach to the ROS Master is used.

Figure 2 gives an overview of some of the essential processes and ROS topics that are present in the HDRC3 architecture. These include functionality for use of individual platforms in collaborative scenarios. The functionality associated with the control layer is encapsulated in the Platform Server and accessed by the two languages FCL and PPCL mentioned earlier.

Black arrows indicate calls between processes. Each gray rectangle contains processes and topics that

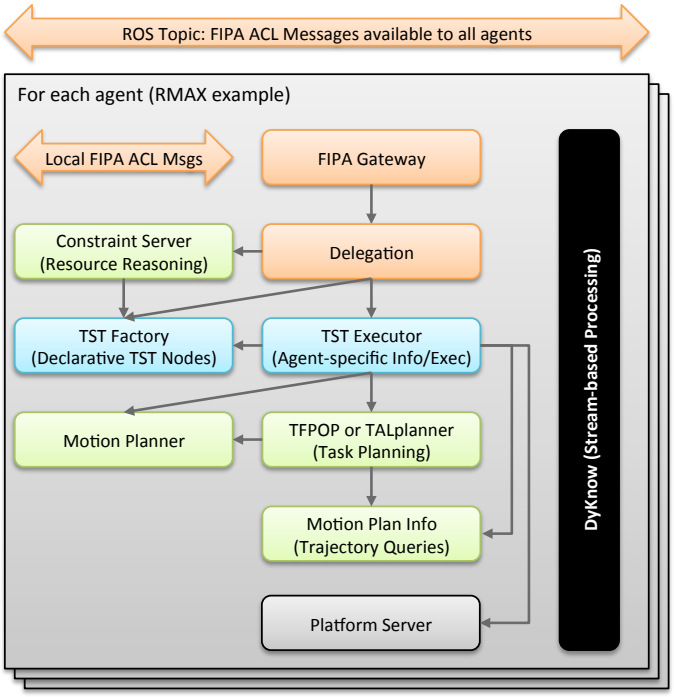


Figure 2: Overview of the ROS-based implementation.

are explicitly associated with a particular agent. Inside this, each rounded rectangle represents a distinct functionality that is currently implemented as one or more related ROS services provided by one or more ROS nodes. Given the fundamental structure of ROS, functionality can easily be moved between nodes without affecting the remainder of the system. Therefore an exact specification at the node level is not relevant at this level of abstraction.

There is one special functionality associated with the HDRC3 architecture which can be viewed as a shared middleware and interfacing mechanism supporting the creation and management of data flow in the architecture. This middleware is called DyKnow and it is based on the abstraction of streams of data.

One of the highly problematic and open research issues in AI and robotics is the large discrepancy between the quantitative and large amount of raw sensor data input to a robotic system and the requirement for sparse, well-structured and grounded qualitative data used by the deliberative functionalities in robotic architectures. This is often called the sense-reasoning gap.

DyKnow is a middleware functionality that has been developed in the HDRC3 architecture which provides a principled means of closing this gap both theoretically and pragmatically. DyKnow provides abstractions for the construction, reasoning, processing and abstraction of data streams within the architecture itself. This middleware is leveraged throughout the architecture and used by many functionalities to great benefit. It is used in a variety of ways at any layer in the HDRC3 architecture and is integrated with ROS.

3.2 Roadmap to the Chapter

Both Figure 1 and Figure 2 provide concise visual depictions of the HDRC3 architecture and its essential functionalities. This framework will be used to structure and provide a roadmap for the remainder of the chapter.

Section 4 describes the UAS Tech Lab (UASTL) RMAX helicopter system which includes a distributed hardware architecture and a suite of sensors integrated with the base Yamaha RMAX helicopter system.

Section 5 describes the Control Layer. This includes a description of the Control Kernel itself with its primary flight modes (Section 5.2), in addition to two complex compound flight modes: a Path Following

Control Mode (Section 5.3) and a Vision Based Landing Mode (Section 5.4).

Section 6 describes the Reactive Layer. This includes a description of Hierarchical Concurrent State Machines and a real-time HCSM interpreter (Section 6.1). This is a generic capability that requires instantiation and use in terms of specific HCSMs. Section 6.1.1 describes these specific HCSMs used in the UASTL RMAX system. The higher-level ROS-based system shown in Figure 2 interfaces to the real-time control functionalities (Section 5.2) through the *Platform Server*. This is described in Section 6.2. Complex tasks and missions are concretely represented as Task Specification Trees using elementary actions such as flying together with a number of task composition constructs such as sequences, concurrent execution, and loops (Section 6.3). Nodes in such trees, which are general and declarative specifications of tasks to perform, are both created by and stored in a *TST Factory*. The *TST Executor* functionality is responsible for generating platform-specific procedural *executors* for any node type that a particular agent supports. Platform-specific constraints on how a certain task can be performed, such as those related to the flight envelope of the RMAX, are also specified here.

Section 7 describes the Navigation Subsystem. This subsystem includes the path and motion planning algorithms (Section 7.1) in addition to the specific HCSMs used to coordinate these processes (Section 7.2).

Section 8 describes DyKnow, the stream-based processing middleware framework used for managing data and knowledge flow within the HDRC3 architecture. This functionality is used by many other functionalities within the architecture. Connections to DyKnow are omitted in Figure 2 in order to avoid clutter, as information processed by DyKnow can be used by any deliberative or reactive functionality. A traffic monitoring example, where DyKnow plays an essential role, is described in Section 8.7.

Section 9 describes many of the high-level deliberative functionalities in the HDRC3 architecture that can be used by other functionalities in the architecture. One open research issue of great importance is the requirement of formally verifying the behavior of complex deliberative functions which are often dependent on rich world models and also highly non-deterministic in nature. The HDRC3 architecture uses temporal logics not only for specification, but for on-line monitoring purposes. Section 9.1 describes Temporal Action Logic (TAL) which is used for formally specifying task plans and Task Specification Trees. Section 9.2 describes a *task planner* called TALplanner that is used in the HDRC3 architecture and is highly integrated with motion planners and execution monitoring mechanisms. One of the responsibilities of the task planner is to expand *goal nodes* in a Task Specification Tree into detailed task specifications, which is why it can be called from the TST Executor shown in Figure 2. The task planner can also call the motion planner and Motion Plan Info functionalities in order to estimate whether a given flight action is feasible as part of a plan being generated, and if so, at what cost. Section 9.3 describes a very sophisticated and efficient execution monitoring functionality based on the use of model checking temporal logical formulas in real-time through use of a progression algorithm. Section 9.4 describes an extension to Temporal Action Logic that allows one to formally specify complex missions. These missions can be compiled into Task Specification Trees and are also useful in specifying collaborative missions.

Section 10 briefly discusses collaborative systems consisting of multiple platforms. In a collaborative setting, *delegation* is the key to assigning the responsibility for specific aspects of a mission to specific agent-based unmanned systems. Delegation requests are specified as speech acts in the FIPA Agent Communication Language (ACL) (Foundation for Intelligent Physical Agents, 2002) and are sent between agents through a set of communication channels, currently implemented using ROS topics. A *gateway* functionality acts as a clearinghouse for all inter-agent communication based on such speech acts. Determining whether an agent can accept a delegation request also requires reasoning about whether and how the agent can satisfy the temporal and resource-related constraints associated with a task. This is implemented through a resource reasoning capability that includes a *constraint server*. These particular functionalities are shown in Figure 2.

Section 11 provides examples of several highly complex mission scenarios that have been used to empirically test the HDRC3 architecture, its various functionalities, and the tight integration required among the functionalities. UASTL RMAXs have been deployed and used in these scenarios and variations of them on

Figure 3: The UASTL RMAX helicopter.

numerous occasions to not only refine the HDRC3 architecture but also to ensure repeatability of missions and robustness of the architecture in real-world environments.

Section 12 summarizes and discusses the generic nature of the architecture and many of its functionalities.

4 Unmanned Aircraft Platforms

The HDRC3 architecture and variations of it have been used on a number of different research platforms, in particular in the context of research with heterogeneous collaborative systems (Doherty et al. (2013); http://www.ida.liu.se/divisions/aiics/aiicssite/). For example, a lightweight version of the architecture has been ported to the LinkQuad, a micro-aerial vehicle platform developed within the UAS Tech Lab (http://www.uastech.com). However, this chapter focuses solely on the use of the HDRC3 architecture on a Yamaha RMAX system which has been extended for autonomous use. This platform is called the UAS Tech Lab (UASTL) RMAX (Conte and Doherty, 2009; Doherty et al., 2004; Rudol and Doherty, 2008; Wzorek et al., 2006a).

4.1 The RMAX Helicopter Platform

The UASTL RMAX platform is a modified Yamaha RMAX helicopter (Figure 3). The Yamaha RMAX helicopter is a radio-operated commercial system marketed in Japan by Yamaha Motor Co. Ltd. Two of these were acquired and used as a basis for developing fully autonomous systems. The HDRC3 architecture is integrated on both systems.

The RMAX is approximately 2.7×0.7 meters with a total length of 3.6 meters including the rotor. It uses a 21 HP two-stroke engine and has an empty weight of 61 kg and a maximum takeoff weight of 95 kg.

4.2 Computational Aspects

The computer systems used on-board the UASTL RMAX need to work reliably in harsh environmental conditions characterized by high vibrations, limited space and a high range of operating temperatures. The PC104 is an industrial grade embedded computer standard that is especially suited for such applications. The on-board system developed for the UASTL RMAX contains three such PC104 embedded computers. These embedded computers physically reside in a specially designed box which is integrated with the RMAX and is shown in Figure 3. Figure 4 provides a hardware schematic of the integrated system.

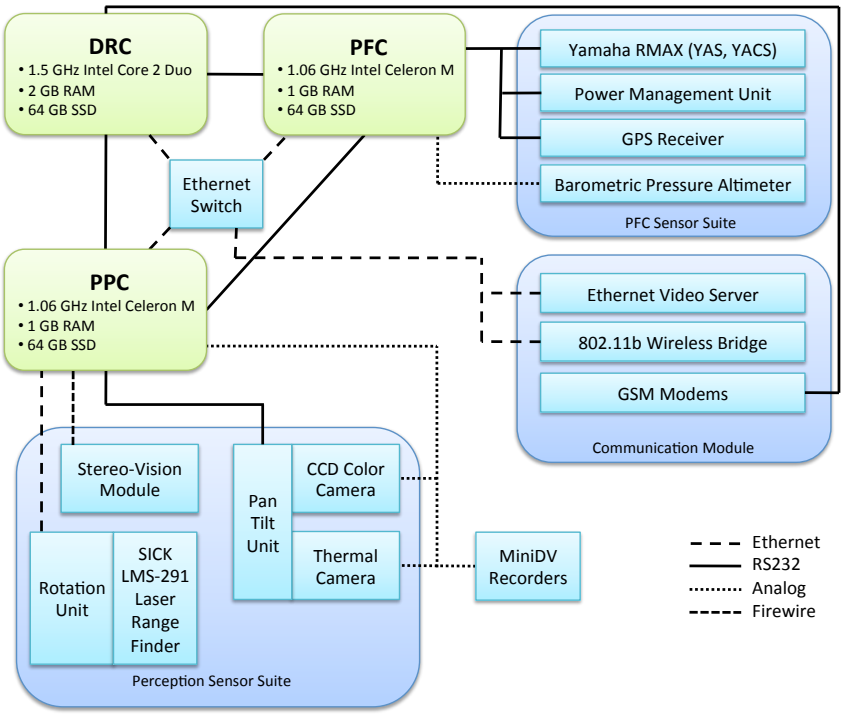


Figure 4: On-board hardware schematic.

- The Primary Flight Computer (PFC, 1.06 GHz Intel Celeron M) handles core flight functionalities and control modes. This computer is directly connected to a GPS receiver and several additional sensors including a barometric altitude sensor. The PFC is also connected to the UASTL RMAX helicopter through the standard Yamaha Attitude Sensor (YAS) and Yamaha Attitude Control System (YACS) interfaces.
- The Deliberative/Reactive Computer (DRC, 1.5 GHz Intel Core 2 Duo) executes all high-level autonomous functionalities such as path planning (Section 7.1), stream-based reasoning (Section 8), and mission planning (Section 9.2).
- The Primary Perception Computer (PPC, 1.06 GHz Intel Celeron M) runs software related to the use of cameras, range finders and similar equipment. This includes image processing and sensor fusion.

Due to the high degree of vibration on-board the UASTL RMAX, in its current configuration it is not possible to use standard hard drives for storage. Instead all computers use Solid State Disk (SSD) technology.

Network communication between the on-board computers is realized with serial lines (RS232C) as well as Ethernet. The serial lines are used to implement a robust real-time communication channel between each of the on-board computers.

4.3 Communications, Sensors and Other Hardware

A wireless Ethernet bridge as well as a GSM modem is available for communication with ground control stations. GSM provides a mature, commercially available communication infrastructure that permits the operation of AVs at large distances, out of sight from the ground operator. It also provides a redundant alternative in the case where other communication frequencies are jammed. In order to test the feasibility of GSM technology for interfacing purposes, a multi-modal graphical user interface has been designed, constructed and implemented on a mobile telephone (Wzorek et al., 2006b). This has been used as a mobile ground operator control system where mission plans can be specified, uploaded and monitored through the mobile telephone.

(a) CCD Block Camera on PTU.

(b) Infrared Camera.

Figure 5: Cameras used on the UASTL RMAX.

A camera platform is suspended under the AV fuselage, vibration-isolated by a system of springs. The platform consists of a Sony FCB-780P CCD block camera (Figure 5(a)) and a ThermalEye-3600AS miniature infrared camera (Figure 5(b)) mounted rigidly on a Pan-Tilt Unit (PTU). Section 11.1.1 presents a method for using both cameras in an algorithm for finding human bodies and building saliency maps. The video footage from both cameras is recorded at full frame rate by two miniDV recorders to allow processing after a mission.

A modified SICK LMS-291 laser range finder has also been integrated with the UASTL RMAX using a rotation mechanism developed in-house. It is mounted on the front as shown in Figure 3. This system provides the ability to generate high-accuracy 3D models of the environment in which the aircraft is operating, without the need for reflectors, markers or scene illumination (Section 11.2).

A software-controlled power management unit provides information about voltage levels and current consumption. It also allows for remote power control where all devices can be switched on or off through software. The unit controls all three PC104 computers, all sensors, and the wireless Ethernet bridge.

The HDRC3 architecture is distributed on this hardware platform. The control layer of this architecture, which in many ways operates close to the hardware, will now be described.

5 The Control Layer

The Control Layer implements basic continuous control modes used by the UASTL RMAX. These control modes are implemented using continuous control laws as a basis. Already at this level, the algorithms used are quite complex. For instance, two essential modes described in this section, path following (Section 5.3) and vision-based landing (Section 5.4), require tight integration with the architecture and involve discrete event control capability (Section 6.1). This section describes both these modes in addition to the control kernel (Section 5.2) and its real-time demands (Section 5.1).

5.1 Real-Time Aspects

Certain functionality, such as control laws in the Control Kernel, requires support for real-time processing with latency guarantees. Since the current implementation of the HDRC3 architecture runs on a Linux-based operating system, the Real-Time Application Interface (RTAI (Mantegazza et al., 2000)) is used for this purpose.

RTAI is a hard real-time extension to the standard Linux kernel and provides industrial-grade real-time operating system functionality. RTAI allows for the creation of a kernel module that takes full control over the CPU and can suspend itself in order to let user-space applications run. Such a module is used to implement real-time functionalities in the HDRC3 architecture. The standard Linux distribution running on the same CPU is viewed as a separate task with lower (non-real-time) priority, running preemptively so that it can be interrupted by real-time processes at any time. Real-time and non-real-time processes can then communicate through shared memory.

All computers in the UASTL RMAX platform (Figure 4) 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 a minimal number of non-real-time user space applications (for example, an SSH daemon for remote login). On the other extreme, the DRC uses the real-time part only for device drivers and real-time communication, while the majority of its time is spent on running deliberative services.

5.2 The Control Kernel

The Control Kernel (CK) encapsulates all core control functionalities, including a set of continuous control laws implementing the basic flight and payload control modes as well as control over the perception capabilities that can be used by the reactive and deliberative layers. It is also responsible for switching between control modes through the use of Hierarchical Concurrent State Machines (Section 6.1).

The kernel is distributed on two on-board computers (PFC and PPC), and also coordinates real-time communication between these computers as well as between CKs of other robotic systems when collaborative missions are involved. However, the kernel itself is self-contained and only the part running on the PFC computer is necessary for maintaining flight capabilities. This enhances the safety of the operation of the UASTL RMAX platform.

The following flight control modes have been implemented on the UASTL RMAX platform:

- **Hovering:** Keep the helicopter in the desired position, heading and altitude given current sensor readings. The mode is implemented using a set of PID loops.
- **Take-off:** Bring the helicopter from the ground to a specified altitude.
- **Path following:** Fly along a specified segmented path (Section 5.3).
- **Vision-based landing:** Land the AV on an artificial pattern. This mode does not require a GPS signal, instead it relies on a more accurate relative state estimation performed on-board (Section 5.4).
- **Reactive car following:** Maintain a specified distance to a moving car. The car position estimation is performed on board the AV and is based on color tracking using camera images.

Additionally, a set of control modes for payloads has been developed:

- **Camera Pan-Tilt absolute/relative control:** Maintain the absolute/relative angles of the pan-tilt mechanism.
- **Camera Pan-Tilt visual servoing control:** Keep a tracked object in the center of the camera image. The position of the object is provided by image processing functionalities such as vision-based color object tracking.
- **Camera Pan-Tilt look at GPS coordinate control:** Keep a geographical location in the center of the camera image.
- **Camera (color, IR) parameter control:** Actively control camera parameters such as shutter, zoom and iris. Among other things, this allows for adjusting to the current lighting conditions.
- **Laser range finder rotational mechanism control:** Actively control parameters of the rotational mechanism, such as rotational speed. Among other things, this allows for adjusting the trade-off between speed and accuracy when building a 3D map.

- **Laser range finder parameter control:** Actively control parameters of the laser range finder, such as its measurement distance resolution and angular resolution. Among other things, this allows for adjusting the trade-off between speed and accuracy when building a 3D map.

Perception functionalities accessible in the control kernel include:

- **Vision-based state estimation:** Provides a very accurate pose estimate in relation to a specifically designed pattern. This functionality is used for the autonomous vision-based landing mode.
- **Vision-based object tracking:** Provides an estimate of the relative position to a moving object based on the color and/or thermal image streams. This is used in the reactive car following control mode.
- **Vision-based human body identification:** Identifies human body positions in a stream of color and thermal images (Section 11.1.1).
- **Laser range finder 3D map building:** Provides a 3D elevation map of the environment based on laser range finder data. The map can be used for various applications such as navigation where it is used as input to a path planner (Section 11.2).
- **Localization based on reference images:** Provides an absolute position estimate in GPS denied environments based on geo-referenced imagery (Conte, 2009).

Two complex flight control modes will now be considered in more detail.

5.3 Path Following Control Mode

One of the control modes executed and coordinated by the Control Kernel is the Path Following Control Mode (PFCM (Conte, 2009; Conte et al., 2004), bottom part of Figure 17 on Page 33) which executes paths consisting of a set of segments. Figure 7 presents an example path consisting of three segments.

Given a specific robotic platform, a classical problem in control theory is to find a trajectory compatible with its kinematic and dynamic constraints. A trajectory is the evolution of the state of a robot. The robot state is a vector composed of a set of time-dependent variables describing its kinematic and dynamic status. In this specific case the most relevant components of the state vector are the three dimensional position, velocity and acceleration, the attitude angles (pitch, roll, yaw) and the attitude rates. In general, the problem of finding a trajectory compatible with the platform dynamic constraints is a very difficult problem. In the context of a robotic helicopter flying in a cluttered environment, the problem becomes even harder since the platform dynamics is unstable. In addition, the flight path must not collide with the obstacles in the environment. The methodology used here to deal with this complexity is to decompose the problem as follows:

- *Decoupling the path-planning problem from the platform dynamics*, by first searching for a collision-free path and then adapting this path to the dynamics of the AV platform. This yields a dramatic dimensionality reduction in the search space. In the HDRC3 architecture, the task of finding a collision-free path is solved by a path planner.
- *Dividing the control problem into fast and slow dynamics.* This is usually achieved using an *inner* feedback loop which has the task of stabilizing the attitude dynamics. The inner loop is usually implemented at a relatively high frequency (around 200 Hz). An *outer* feedback loop is then used to stabilize the platform position and velocity at a lower frequency (around 20 Hz). Dividing the problem in this way facilitates analyzing and solving the platform control task. In other words, the inner loop takes care of stabilizing and controlling the rotational dynamics while the outer loop takes care of stabilizing and controlling the translational dynamics. They are implemented at different frequencies because the platform rotational and translational dynamics have different time constants. The rotational dynamics is usually faster than the translational dynamics.

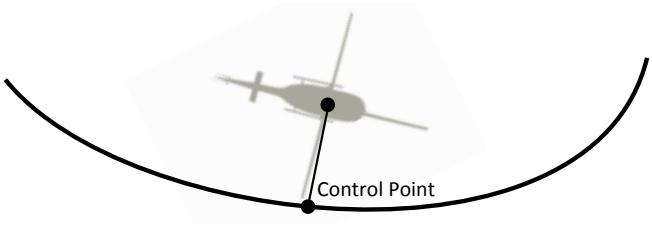


Figure 6: Control point on the reference path.

This section deals with the problem of finding a *guidance* algorithm which enables the helicopter to follow a geometric path generated by a path planner. The distinction between path and trajectory must be emphasized here. As previously mentioned, a trajectory is the evolution of the state of the vehicle in the state-space. The trajectory is usually time-dependent. The path is a geometric description of the flight course and it is time-independent. The problem addressed here is referred to in the literature as the *path following* problem. The approach is suitable when a geometrical description of the path is provided in a parameterized polynomial form. In this case the so-called virtual leader approach (Egerstedt et al., 2001) can be applied.

By using the path following method, the helicopter is forced to fly close to the geometric path with a specified forward speed. In other words, following the path is always prioritized over following other trajectory parameters as for instance the desired speed. This is a requirement for robots that, for example, have to follow roads and avoid collisions with buildings. The PFCM method developed is weakly model-dependent and computationally efficient.

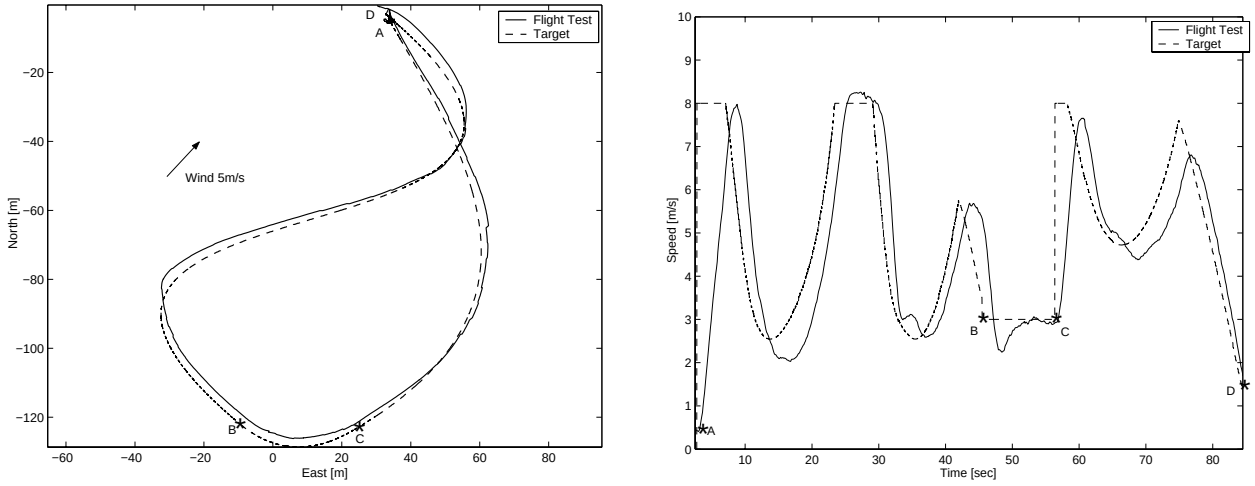
The approach adopted here uses a guidance algorithm similar to the one described in Egerstedt et al. (2001) which is also known as virtual leader. In this case the motion of the control point on the desired path is governed by a differential equation containing error feedback which gives great robustness to the guidance method. The control point (Figure 6) is basically the closest point on the path relative to the current platform position.

A path is composed of several consecutive segments. Each segment is mathematically described by the equation $\vec{p}(s) = As^3 + Bs^2 + Cs + D$, where A, B, C and D are 3D vectors calculated from the boundary conditions of the segment. s is the segment parameter and assumes values between 0 (start of the path) and 1 (end of the path). The control point is found using a feedback method. Details of the algorithm can be found in Conte (2009); Conte et al. (2004). Once the control point is found, from the path equation it is possible to compute all the geometric parameters which are then used for control purposes. The parameters are the 3D position, 3D path tangent and 3D path curvature.

Path segments are generated in the deliberative layer of the HDRC3 architecture by the path planner functionality (Section 7). The segments in a path are passed sequentially to the PFCM for execution. Together with the path parameters, the path planner provides a desired target velocity to the PFCM. During execution, the PFCM performs a compatibility check between the desired target velocity and the maximum allowable velocity at that precise point on the path. The maximum allowable velocity profile for a path is calculated using the path curvature R and the maximum helicopter roll angle ϕ_{max}: $V = \sqrt{|\phi_{max} g R|}$, where g is the gravity constant. In addition, the path curvature is used as a feed-forward term in the outer control loop in order to reduce the tracking error. Besides the basic control task, the PFCM returns a set of status flags which are used to coordinate the path segment switching mechanism.

A safety braking procedure is activated in case the next segment is not provided by the navigation subsystem (for example due to communication failure) before a specific point in time. This time point is calculated using the minimum distance necessary to safely stop the helicopter at the end of the current segment.

Figure 7 shows a multi-segment path and the relative velocity profile during a real flight-test using the UASTL RMAX helicopter. The segments AB, BC and CD have the desired target velocity set at 8 m/s, 3 m/s and 8 m/s, respectively. One can observe how the PFCM adjusts the target speed, slowing down the



(a) Target and executed path.

(b) Velocity profile.

Figure 7: Flight-test of a multi-segment path.

helicopter in order to meet the compatibility requirements discussed above.

5.4 Vision-Based Landing Mode

Many autonomous landing systems for AVs are based on GPS and a dedicated close range sensor for accurate altitude measurement (radar altimeter, sonar, infrared or theodolites). However, in urban environments, buildings and other obstacles disturb the GPS signal and can even cause loss of signal (multi-path effects, EM noise due to active emitters). Once the GPS signal is lost, the dead reckoning capability of affordable on-board inertial navigation systems does not allow precision navigation for more than a few seconds.

In contrast, the vision-based landing mode discussed here is self-contained and not jammable (Merz et al., 2004). In the current implementation it provides a position measurement one order of magnitude more accurate than standard GPS (cm accuracy or better). Additionally, it does not require a complex infrastructure, only a pattern placed on the designated landing area (Figure 8). To improve robustness, readings from an inertial measurement unit are fused with the position sensed by the vision system.

The landing system includes three main components: (a) a vision system that allows robust pose estimation from a suitable distance at a sufficient rate with low latency, using a landing pad of minimal size; (b) an algorithm to fuse vision data with inertial measurements, and; (c) a matching control strategy.

Vision System. The vision system consists of a monocular camera mounted on a pan/tilt unit (PTU) and a landing pad (a foldable rectangular plate) with a reference pattern on its surface (Figure 8). The reference pattern is designed specifically for fast recognition, accurate pose estimation for close and distant range, minimum size, and minimal asymmetry. The choice of black circles on a white background allows for fast detection and provides accurate image features. From the projection of three circles lying on the corner points of an equilateral triangle, the pose of an object is uniquely determined assuming all intrinsic camera parameters are known. Circles are projected as ellipses, described by the center point, the semi-major axis, the semi-minor axis, and the semi-major axis angle. The pose of the landing pad with respect to the camera coordinate system is estimated by minimizing the reprojection error of the extracted center points and semi-axes of the three ellipses. Five circle triplets of different size are used (radius: 2 to 32 cm, distance: 8 to 128 cm) with a common center point to achieve a wide range of possible camera positions. Each triplet is uniquely determined by a combination of differently sized inner circles.

The output of the image processing unit is the camera pose relative to the pattern. The pose parameters are converted to helicopter position and attitude using angles from the PTU and known frame offsets and rotations.

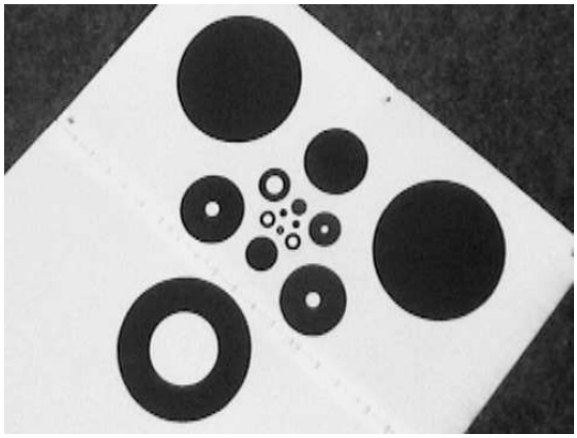


Figure 8: Landing pad with reference pattern seen from the on-board camera.

Sensor Fusion. The position and attitude estimates provided by the vision system cannot be fed directly into the controller due to their intrinsic lack of robustness: the field of view can be temporarily occluded (for example by the landing gear), the illumination conditions can change dramatically just by moving a few meters (sun reflections, shades, etc.). On the other hand, vision readings are very accurate, when available.

Hence, a navigation filter based on a Kalman filter (KF) has been developed, fusing highly accurate 3D position estimates from the vision system with inertial data provided by the on-board accelerometers and angular rate gyros. Besides filtering out a large part of the noise and outliers, the filter provides an adequate dead reckoning capability, sufficient to complete the landing even when the vision system is "blind" due to occlusions between the camera and the pattern (landing gears) or due to sun reflections.

Control Strategy. The requirements set on the flight control system during landing are the following:

1. It should be possible to engage the landing mode from any point where the landing pad is visible, meaning approximately within a 20 m radius hemisphere centered on the pattern.
2. Once the landing mode is engaged, the helicopter state should be compatible with the proper functionality of the vision system until touchdown. Thus, during the approach phase the following should be considered: (a) the helicopter's position and attitude should not cause physical occlusion of the visual field; (b) the regions where the accuracy of the vision system is worst should be avoided, if possible; (c) the helicopter velocity and angular rates should not saturate the pan/tilt unit's capability for compensation: too high angular rates of the visual beam may result in blurred images; (d) the position of the dominant light source (sun) should be considered, to avoid full reflections.
3. The wind direction has to be taken into account: tailwind landings should be avoided.
4. The control system should be dimensioned for wind levels up to 10 m/s.
5. The engine should be shut down autonomously once touch-down is detected. Detection should be timely: Early detections cause high touchdown loads and late detections can cause ground resonance.
6. The vertical velocity at touchdown should be of the same order of magnitude as for a manual landing.

Experimental Results. Numerous autonomous landings were conducted from different relative positions to the landing pad within the specified envelope, on grass and snow fields, with different wind and illumination conditions. The vertical velocity at touchdown ranged between 18 and 35 cm/s, corresponding to load factors of about 1.4 g on grass fields. The horizontal velocity was in the order of magnitude of 15 cm/s. The average touchdown point precision was about 42 cm (13% of the rotor diameter).

This section has isolated and described the essential aspects of the control kernel and two complex control modes. In order to put these to use, though, they have to be integrated and interfaced with many different parts of the HDRC3 architecture. These details are described in Section 6 and in Section 7.

6 The Reactive Layer

The functionality associated with the reactive layer is an essential part of the HDRC3 architecture since it provides transitional capability between functionality in the deliberative layer and functionality in the control layer. The tasks associated with this layer range from very low-level state machine based processes which provide discrete control capability for the control layer to much higher level procedural mechanisms which provide discrete control capability for complex functionalities in the deliberative layer. In fact, the abstractions and language interfaces used for these transitions are what make such architectures so unique and useful. Specification and design of these mechanisms involves more than just engineering skills.

There is an independent generic flavor to the mechanisms required for combining both discrete and continuous control capabilities and managing the processes evoked by task descriptions at many different levels of abstraction with required temporal latencies to guarantee quality of service and robustness.

This section will describe three generic interfacing capabilities associated with the HDRC3 architecture in addition to several task execution mechanisms for tasks defined at different levels of abstraction. Section 6.1 considers the specification of complex combinations of state machines using Hierarchical Concurrent State Machines as a specification language. Additionally, an algorithm for executing HCSMs in real-time using an on-line interpreter is described. Section 6.2 describes the two abstract languages FCL and PPCL and a Platform Server which relates these languages to combinations of low-level control modes. This mechanism provides a high-level abstraction for any functionality in the system to access the control layer and combinations of control modes in an efficient and declarative manner. Section 6.3 describe Task Specification Trees. TSTs provide both a declarative means of specifying complex tasks and an execution mechanism which allows for distributed execution of TSTs.

6.1 Hierarchical Concurrent State Machines

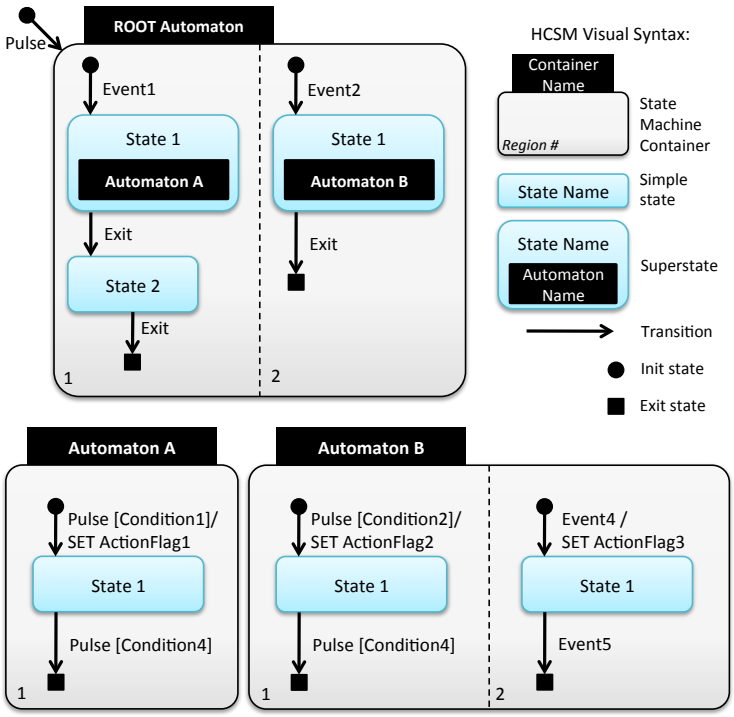
Missions generally involve multiple control modes used in sequence or in some cases in parallel. For example, in building surveillance one would use take-off, hovering, path following, camera control, landing, and possibly other control modes. Even if only a sequence of control modes is needed, execution is non-trivial as it requires a smooth handover between the control mode that is terminated and the one that is started, leading to what is often called the *mode switching problem*.

The conditions for being able to switch to a new control mode vary depending on the particular mode transition. For example, in the case of switching between hovering and path following modes, one must verify that the current helicopter heading is aligned with the path to be flown. If the heading difference is too large, an additional yaw maneuver is necessary before the transition to the path following mode. Otherwise the execution of the path could result in a maneuver that potentially leads to a crash.

Solutions to the mode switching problem are often based on Finite State Machines (FSMs) (Harel, 1987; Koo et al., 1998), a mathematical abstraction used to create a behavior model of a system. FSMs are composed of a finite number of states, state transitions that can be guarded by conditions, and actions. For example, a condition can depend on the evaluation of a sensor value. The input to a state machine is a sequence of symbols, events or commands. Standard FSMs such as Moore (Moore, 1956) and Mealy (Mealy, 1955) machines have been successfully used in the past for modeling systems for various purposes. However, several extensions are necessary to make them useful for modeling complex real-time systems.

One major problem related to FSMs is their flat single-level structure, which provides no easy way to partition a problem into sub-problems that can be dealt with separately. Each state has to be modeled explicitly, leading in the worst case to a combinatorial explosion in the number of states. This results in large and unmanageable models even for moderately complex systems.

It would be much easier to define a system gradually with different levels of granularity or with the help of various levels of abstraction. An obvious additional advantage would be the reusability of existing modeled parts of the system in different contexts. A novel modeling and development framework for hybrid



Syntax of a transition in EBNF notation:
Transition = EventID ["[" Guard "]"] ["/" ActionList]
ActionList = Action {"," Action}
Action = SET FlagID | UNSET FlagID | GET MemorySlotID | PUT MemorySlotID |
SEND EventID MemorySlotID DestinationID
Guard = ["¬"] FlagID | Guard Op Guard | "(" Guard ")"
Op = "∧" | "∨"
(* all *ID* symbols are ANSI C id-tokens *)
(* all *Name* symbols are sequences of ASCII characters *)

Figure 9: HCSM visual syntax with an example of three state machines.

control systems called *hierarchical concurrent state machines* (HCSMs (Merz, 2004; Wzorek, 2011)) has been developed. HCSMs are based on a combination of Moore and Mealy machines with several extensions which include support for hierarchy and concurrency. This allows all low-level components to be efficiently modeled and contributes to a solution to the mode switching problem in the HDRC3 architecture. It permits all *functional units* of the control system to be coordinated ranging from the lowest level (such as device drivers), through the use of control laws (such as hovering and dynamic path following) and communication, to high-level deliberative components. The system has proven to be robust, reliable and easy to extend and has been used in a number of autonomous missions during a period of several years.

The HCSM specification language and computation model are to an extent influenced by the Statecharts formalism (Harel, 1987) which has certain similarities to MATLAB Stateflow. However, the HCSM language and model have several differences intended to support a clean visual syntax, a clear semantics, and a highly efficient implementation providing strong real-time guarantees. HCSMs can also be interpreted as opposed to being compiled. This facilitates reconfiguring the system at run-time.

State Machines, Hierarchy and Concurrency. A *state machine* consists of states and transitions. A state represents any activity of a system at any level of abstraction. Two main types of states are defined: *simple states* and *superstates*. A superstate represents a nested state machine, allowing a hierarchy of state machines to be created. Two special types of simple states are defined: *init* (the starting state) and *exit* (which terminates execution). Figure 9 presents a visual syntax for HCSMs and provides an example of a simple state and a superstate: *State 2* and *State 1* of the root automaton, respectively. In the remainder of this chapter the terms *state machine* and *automaton* will be used interchangeably and will refer to HCSMs.

A *state machine container* is a collection of one or more concurrent (child) state machines. Each of

these machines is contained in a *region*. Regions are ordered by a consecutive number (RegionNumber) and are separated by a dashed line in the visual syntax. For example, *Automaton B* contains two regions in Figure 9.

Hierarchies provide a powerful mechanism for encapsulating specific behaviors. The framework permits reuse of state machine containers, providing all the necessary means to execute multiple instances of a state machine. Such encapsulation also provides a practical way of aborting a particular behavior. *Concurrency* prevents combinatorial explosion of states, which would occur in FSMs, and permits easy handling of asynchronous state machines. In the HCSM visual language, a *hierarchy* is modelled by vertical state machine decomposition and *concurrency* by horizontal decomposition (Figure 9).

An example where one can take advantage of concurrency and hierarchy is when modeling complex behaviors such as vision-based landing (Section 5.4). The landing mode consists of several lower-level behaviors controlled by the main superstate of the landing mode. For instance, it includes control laws steering the helicopter and it coordinates the camera system and image processing functionalities. When the landing behavior is activated, several state machines modeling the necessary activities are executed. These include searching for a pre-defined pattern with the camera system and inputing the image processing results to a Kalman filter which fuses them with inertial measurements. Once the pattern is found another state machine controls the camera in order to keep the pattern in the center of the image. This increases the robustness of image processing when the helicopter is close to the ground or in the presence of strong wind gusts.

State Transitions, Events and Guards. A transition between two states is triggered by an *event*. Events in the HCSM framework can be generated internally by the state machine itself or externally. Both asynchronous (or sporadic) and periodic events are supported.

There are two types of special events. A *pulse* event is a periodic event generated before each iteration of the HCSM algorithm, similar to a clock pulse (discussed later). For example, this can be used to trigger a transition without a specific asynchronous event. An *exit* event is created when a state machine is in its exit state and it is only sent to the parent superstate informing it that a child automaton has finished its execution.

State transitions can optionally be guarded by *conditions* in the form of Boolean expressions. If an event for a particular state transition has been generated and the condition guarding the transition is TRUE, then the state transition takes place.

Activities vs. Actions. As in Harel (Harel, 1987), a distinction is made between actions and activities. Actions have no duration (zero-time assumption) and are executed in transitions (as in a Mealy machine), while activities take time and are associated with states (as in a Moore machine).

During each transition, a possibly empty set of *actions* can be executed. Supported actions include setting binary flags (SET *flag-name*), sending events (SEND *event-name data target-computer*), retrieving (GET *source*) and storing (PUT *dest*) data. Data is transmitted using a predefined memory bank with labeled slots. Data from received events is automatically stored in the *cache* memory slot. A GET action copies data from a source slot to the cache slot. A PUT action copies data from the cache slot to a destination slot.

Activities are defined in terms of regularly executed functions. They are coordinated by a set of binary flags which are changed by actions. Functions are executed outside the state machine algorithm and their execution is discussed in the next section.

HCSM Design and Execution. As illustrated in Figure 10, the design of a system in the HCSM framework starts either by generating a visual description of state machines using a graphical tool or (in simpler cases) by directly describing state machines in a text-based language. In either case, tables describing transitions can be derived and passed to the system and are then interpreted by HCSM Interpreters at run-time on the robotic platform. Due to its compact and efficient implementation, the interpreter can run in the real-time part of the system as a periodic task with high execution rate.

The HCSM supports AND/OR/NOT operators in condition statements directly in the visual description and HCSM text files. However, the atomic conditions and the activities used are implemented procedurally. A library of such conditions and activities is generally written in an early development phase, after which

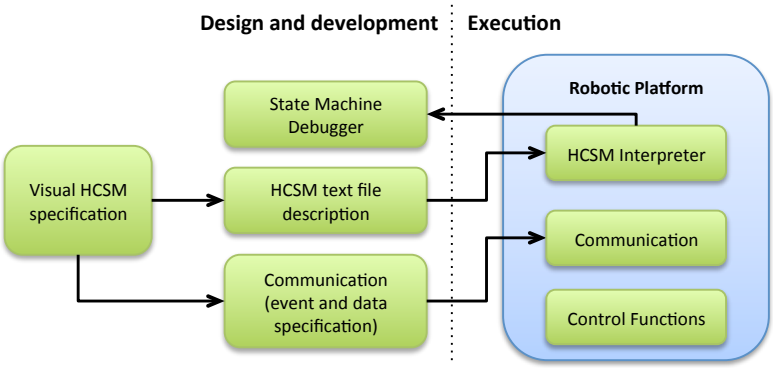


Figure 10: Overview of the HCSM design process and execution.

Algorithm 1 Skeleton of a procedure for execution of a HCSM-based system on a single computer.

– *Main Execution Loop*

1: **while** system is running **do**
2: ...
3: **Communicate();** {Send and receive data packets containing debug information and external events. Events are put in the *external event queue*.}
4: **ExecuteStateMachine();** {Execute algorithm 2}
5: **RunControlFunctions();** {Run the appropriate control functions based on the binary flags set by actions in the HCSM.}
6: ...
7: **end while**

the modeling of the system behavior itself can take full advantage of the flexibility of the interpreted state machine language with no need for recompilation.

Three forms of communication are supported: (1) between states of the HCSM language processed by the same interpreter; (2) between computer systems of the same robot; and (3) between different robots or robots and operators. The first form is realized by an *internal event queue*, while the remaining two are realized by transmitting external events in packets with predefined sizes. Received external events are put in an *external event queue*. The framework directly supports real-time transmissions with built-in integrity checking given that the network satisfies real-time properties, and it automatically generates the required communication code during the design process (Figure 10).

Abstractly, HCSM execution proceeds as shown in Algorithm 1 (see Section 6.1.1 for concrete examples). The automatically generated communication functions are called in the *Communicate* function. Events received by the system are parsed, checked for integrity and put in the external event queue. The HCSM is then executed (*ExecuteStateMachine*). Binary flags set by the state machine transitions (using actions) coordinate the execution of the control functions (*RunControlFunctions*) which are defined by the user.

Details are shown in Algorithm 2. Transitions in HCSMs are divided into *macro steps* and *micro steps*, where external events are considered in the macro step and internal events in the micro step. A macro step (*ExecuteStateMachine* lines 5 to 9) begins by processing the first event from an external event queue. Events are processed in the order they were received. An event can trigger one or more transitions. These transitions may generate both internal and external events which in turn trigger more transitions. The macro step is finished when the external event queue is empty and no more transitions are made. Micro steps are steps within a macro step (*ExecuteStateMachine* lines 7 to 8).

In case the external event queue is empty only the micro steps are executed, which results in all of the internal events being processed (at the beginning one pulse event is in the internal event queue).

The system assumes the "synchrony hypothesis": During a macro step, inputs do not change and exter-

Algorithm 2 The HCSM Algorithm.

– *ExecuteStateMachine*

```
 1: lock memory
 2: create empty internal event queue
 3: append pulse event to internal event queue
 4: repeat
 5:     remove first event from external event queue and append to internal event queue
 6:     while internal event queue not empty do
 7:         remove first event e from internal event queue
 8:         call MakeTransitions(1, e)
 9:     end while
10: until external event queue empty
11: unlock memory
```

– *MakeTransitions(MachineLevel, Event)*

```
 1: for all concurrent state machines M at MachineLevel do
 2:     if Event is received by transition of current state in M and Guards are TRUE then
 3:         call ExecuteActions(actions associated with the transition)
 4:         make transition
 5:     else if current state is superstate then
 6:         call MakeTransitions(MachineLevel+1, Event)
 7:     end if
 8: end for
```

– *ExecuteActions(ActionList)*

```
 1: for each action Action in ordered ActionList do
 2:     if Action is send event action then
 3:         if destinationID is external computer then
 4:             append Event to external communication queue
 5:         else
 6:             append Event to internal event queue
 7:         end if
 8:     else
 9:         execute action
10:     end if
11: end for
```

nal events are not received. In practice, external events are not processed as soon as they arrive but they are buffered until the state machine interpreter is called.

On the UASTL RMAX the state machine interpreter is called periodically every 20 ms in the real-time environment. The duration of one *macro step* is set to 200 μs which corresponds to the worst case execution time. This time depends on the complexity of the state machine being executed, which is expressed in terms of the number of regions and the maximum number of *micro steps* (generated events). The periodic update time and the duration is user-configurable and was empirically chosen for the UASTL RMAX system.

Related work. Many specification languages, software frameworks, and design tools are used in control or embedded system design, including Ptolemy II (Eker et al., 2003), Esterel (Berry, 1992), Statecharts (Harel, 1987), Stateflow (http://en.wikipedia.org/wiki/Stateflow), Simulink (http://en.wikipedia.org/wiki/Simulink), and UML 2 (Booch et al., 2005), but none of these are optimized for building control systems for autonomous robots.

HCSMs are primarily influenced by Harel's Statecharts formalism. State machine based approaches have already been used successfully in many robotic systems. Brooks (1989) for instance uses state ma-

chines to build reactive systems and Kleinehagenbrock et al. (2004) include them in a deliberative/reactive system. Albus and Proctor (2000) propose an architecture for intelligent hybrid control systems which has some similarities with the HCSM framework. It also includes state machines, defines a hierarchy of functional modules and includes a communication system, but it lacks some of the features mentioned above. The HCSM framework supports the component-based design methodology. In Brooks et al. (2005) a component-based framework is proposed which aims for similar goals but it also does not provide some of the features mentioned above (such as real-time aspects). Cremer et al. (1995) propose a framework for modeling reactive system behavior in virtual environment applications. Although some similarities in the modeling formalism exist it lacks some of the fundamental features required for use in robotics applications such as strong real-time execution guarantees and a clear definition of the visual syntax necessary in the design of complex control systems.

6.1.1 Using HCSMs on the UAS Tech Lab RMAX

HCSMs are used on the UASTL RMAX platform as a low-level real-time mechanism for modeling system behavior and are executed on all three of its on-board computers. This is illustrated in Figure 11 which also shows the Platform Server that will be discussed in Section 6.2.1. HCSMs are used on the PFC computer for modeling and executing the control system. HCSMs control a set of sensors and available perception functionalities on the PPC computer. On the DRC computer, HCSMs provide an interface to flight control modes, payload and perception functionalities in the Control Kernel (Section 5) accessible to the high-level deliberative and reactive algorithms. Because the HCSMs running on different computers communicate with each other, all of the individual events used in the system have globally unique IDs.

Several non-real-time processes also run on each of the computers and communicate with the real-time HCSMs through shared memory. Specifically:

- PFC: The TCP/UCP communication handling necessary for sending HCSMs status data used for state machine debugging interfaces (see Figure 10).
- DRC: The *Platform Server*, which provides the deliberative and reactive services with an interface to the Control Kernel through the FCL and PPCL languages (Sections 6.2.1 to 6.2.3).
- PPC: The perception functionalities available in the Control Kernel.

The remainder of the section presents an example using HCSMs running on the PFC computer. An overview of all HCSMs running on the PFC system is given, followed by a description of two state machines involved in control mode switching. A more detailed example of the use of HCSMs in the context of the path following control mode (PFCM) is presented in Section 7.2.

Overview of the PFC HCSMs. Figure 12 presents a hierarchical view of all of the 15 automata that run on the PFC computer. The whole UASTL RMAX system uses 207 events in total. Note that the HCSMs presented in this section are examples from the deployed system. Various extensions and different ways of modeling to achieve the same functionality of the system are of course possible.

The software-controlled power system (Section 4.3) is managed by the PFC computer. This allows all devices except the PFC itself to be switched on and off through software coordinated by the *Power Switch* automaton. At system start-up the *Root* automaton makes sure that at least the DRC computer is switched on (in case it has not been manually switched on by the user). The *Root* automaton also contains a superstate for the *Run* automaton, which in turn contains only one superstate for the following automata: *Power Switch, Sync, Sync DRC, Auto Switch, Control Mode, Visual Navigation* and *User Interface*.

The *Sync* and *Sync DRC* automata are responsible for achieving a common state among the different computers through synchronization. For example, the common state includes the local computer time and the ground position and altitude reference. The *Sync DRC* automaton handles the synchronization between the PPC and DRC computers. The *Sync* automaton is responsible for sending the common state from the

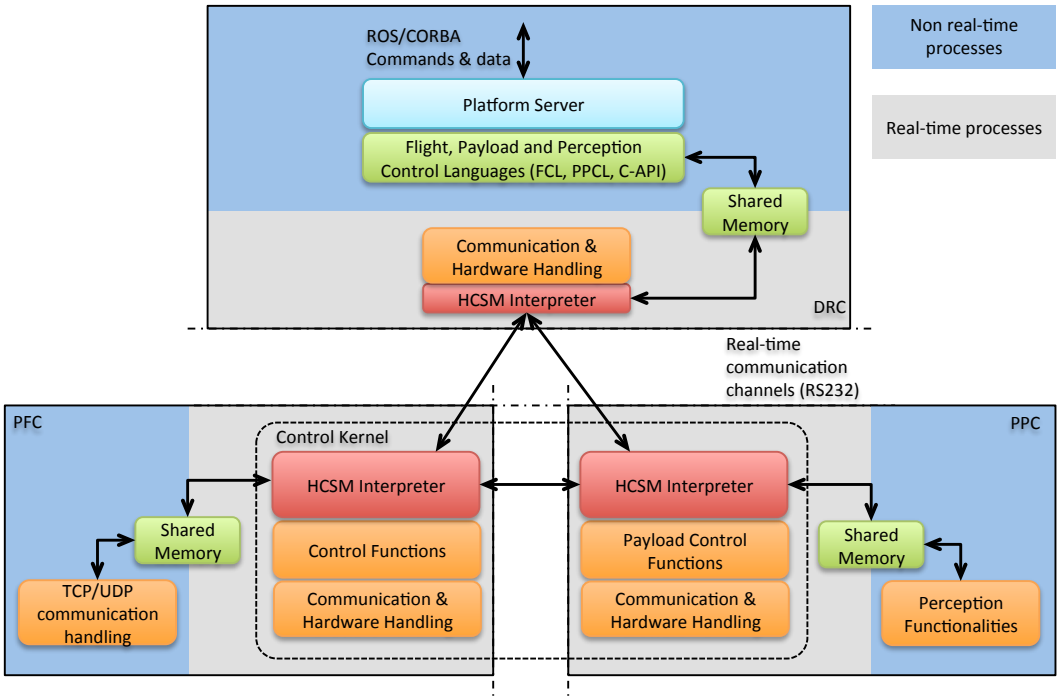


Figure 11: HCSMs and the Control Kernel.

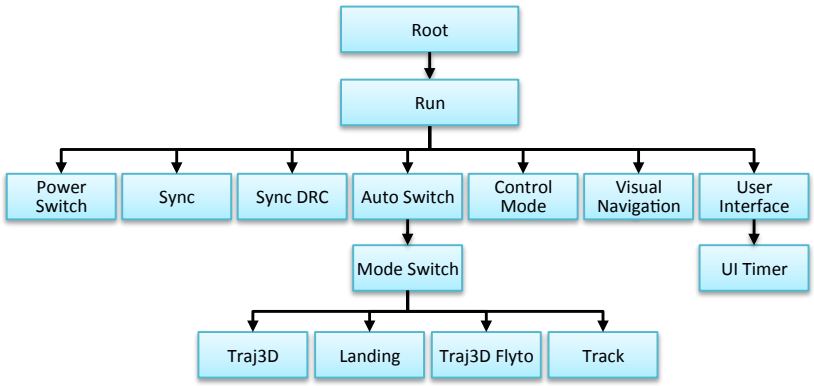


Figure 12: A hierarchical view of the HCSM automata running on the PFC computer.

PFC to the PPC and DRC computers.

The *User Interface* automaton handles commands received from a ground control user interface (UI), while the *UI Timer* automaton implements timeouts for accepting commands by the *User Interface* state machine when using a miniaturized external keyboard.

The *Visual Navigation* automaton coordinates the execution of a localization algorithm that runs on the PPC and is based on reference images (Conte, 2009). The *Auto Switch* automaton handles helicopter platform initialization and operational modes, including the following aspects:

- Initializing the UASTL RMAX helicopter system before it is ready to accept commands. This includes monitoring the engine-on status and the RC radio transmitter status.
- Monitoring sensor data and initializing a Kalman filter used for the state estimation of the helicopter.
- Handling the simulation mode of a hardware-in-the-loop simulator that uses a dynamic helicopter model (Conte, 2007; Duranti and Conte, 2007) and runs all of the software components used during a real flight onboard the AV. This allows newly developed functionalities to be tested before a real flight is performed.

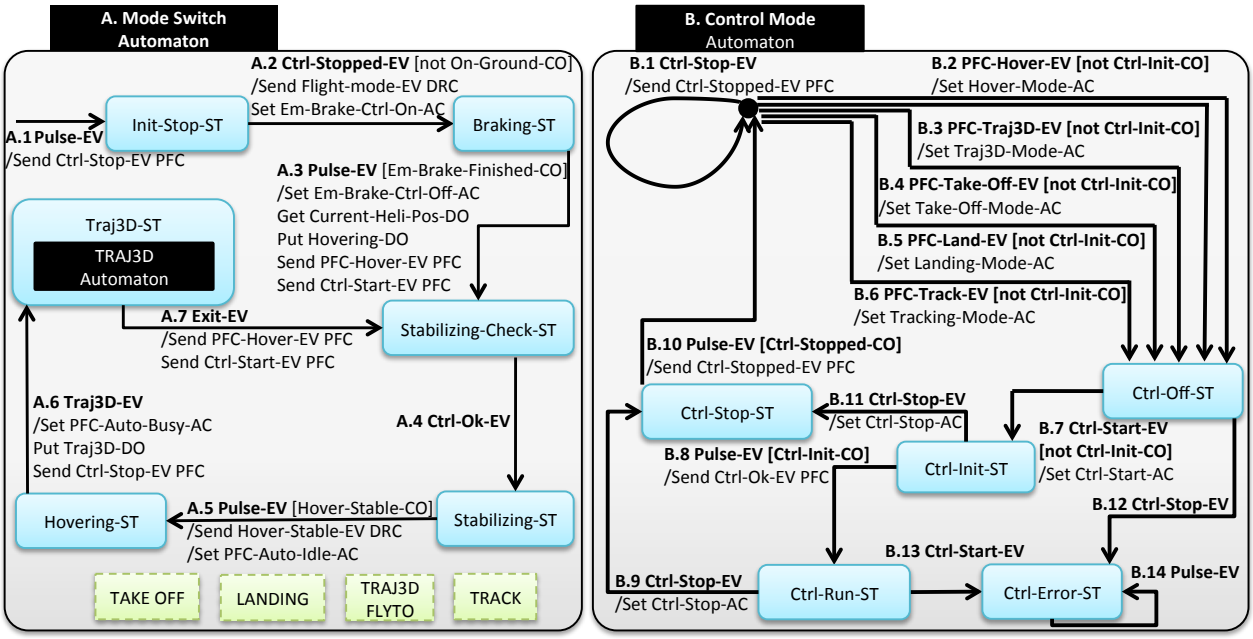


Figure 13: The *Mode Switch* and *Control Mode* automata.

- Handling the RC radio transmitter switch that selects manual or autonomous flight mode.

The *Auto Switch* automaton starts up the execution of the *Mode Switch* state machine after a successful initialization of the system when the autonomous flight mode switch is selected. The *Mode Switch* and *Control Mode* automata in turn handle the switching between flight control modes. This includes initializating, running and handling error conditions for particular control functions.

Certain control modes are also associated with their own mode-specific state machines: *Traj3D* for the path following control mode (Conte et al., 2004), *Landing* for the vision-based landing control mode (Merz et al., 2004), *Traj3D Flyto* for a simplified path following mode which only uses straight line paths and *Track* for a reactive car tracking mode. Other control modes such as take-off and hovering do not require multiple states and are therefore modeled directly by the *Mode Switch* and *Control Mode* automata without any additional state machines.

Periodic Execution. The *Main Execution Loop* in Algorithm 1 executes periodically at a rate of 50 Hz. In each iteration data packets can be sent and received and HCSMs are executed (Algorithm 2) and can update a set of action flags using SET. The active control mode is then updated based on the current values of these flags, after which the flags are automatically unset. Thus, action flags should essentially be seen as triggers that switch control modes. Finally, the currently selected control mode is executed.

Examples. Figure 13 presents two state machines that handle the execution of control modes and implement a mode switching mechanism for the UASTL RMAX. For clarity the following notation is used:

- For fast identification, each automaton has a letter label and all state transitions are numbered.
- Names of states end with “-ST", while names of events end with “-EV".
- Names of conditions end with “-CO".
- Names of SET action flags end with “-AC".
- Names of labeled memory slots (data offsets) end with “-DO".

Each control function follows a predefined template with at least three internal states (*initializing*, *running* and *stopping*) and two internal flags: An input flag used to switch between the internal states and a status flag used for error handling (for example when the initialization of the function fails). Both flags are used

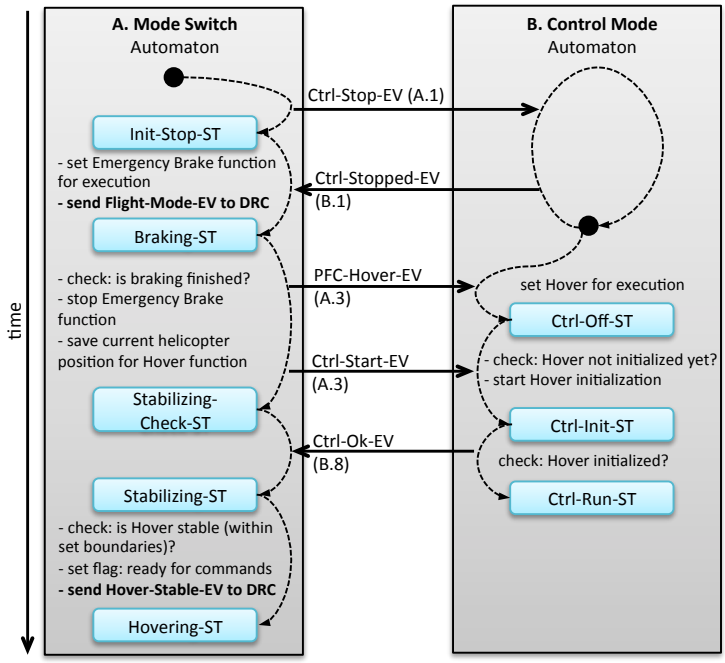


Figure 14: Execution trace: The interaction between the *Control Mode* and *Mode Switch* automata.

by the *Control Mode* automaton which models the internal states of the function (Ctrl-Init-ST, Ctrl-Run-ST and Ctrl-Stop-ST). The additional states present in the automaton (Ctrl-Off-ST and Ctrl-Error-ST) are used to make sure the previous function has been properly stopped and no other functions are initialized.

The *Control Mode* automaton (Figure 13) sets the appropriate input flag and keeps track of the control function status. Because each control function follows the predefined template, the *Control Mode* state machine handles all modes transparently and there is no mode-specific state. When a particular control function should be executed, a state machine (such as *Mode Switch*) sends a mode specific start event (such as PFC-Hover-EV) to the *Control Mode* automaton. This triggers one of the transitions B.2, B.3, B.4, B.5, or B.6 during which appropriate internal flags are set by executing SET actions (for example, SET Hover-Mode-AC). In the next iteration of the *Main Execution Loop* the control function starts its execution passing through the initialization state (Ctrl-Init-ST). If the initialization is successful, the *Control Mode* automaton switches its state to Ctrl-Run-ST and the control function is executed periodically in each iteration.

Control Mode mainly interacts with the *Mode Switch* state machine which implements flight mode switching, including sequentialization and coordination of control function execution. For example, it ensures that after the execution of the path following control mode, a default hovering mode is switched on.

The *Mode Switch* state machine, additionally, generates events that are sent to the high-level system (DRC), for example in the A.2 transition. It also reacts to events sent from the DRC, such as A.6.

Use Case: Example of Control Mode Switching. The following example relates to a simple use case where a backup pilot performs a manual take-off procedure. When the maneuver is finished, the AV control is switched to autonomous mode by using the auto switch button on the RC radio transmitter. A default hovering function should then be engaged as described below.

Figure 14 shows the interaction between the *Control Mode* and *Mode Switch* automata for this example. The time-line shows state transitions and events exchanged between the two state machines. The execution starts with an exchange of two events, Ctrl-Stop-EV (A.1) and Ctrl-Stopped-EV (B.1), to make sure no other control function is active. The *Mode Switch* automaton executes a braking procedure (Braking-ST). This procedure, also called emergency braking, is designed to stop the helicopter before the default hovering mode is engaged. The procedure uses the path following control mode with a predefined straight line path and a zero target velocity value as an input.

When braking has finished the current helicopter position is saved to be used for the hovering function (in the Hovering-DO memory slot). Two events are also sent to the *Control Mode* automaton. The first event (PFC-Hover-EV, A.3) makes *Control Mode* select the hovering function for execution (SET Hover-Mode-AC). The second one (Ctrl-Start-EV, A.3) starts the initialization of the hovering control function.

After a successful initialization, Ctrl-Ok-EV (B.8) is generated and *Mode Switch* changes its state to Stabilizing-ST. The automaton remains in this state until a Hover-Stable-CO condition is satisfied. The condition checks if the position, altitude, heading and velocity are within hovering tolerance bounds. The tolerances used in the UASTL RMAX system are set to 5 m for position, 2 m for altitude, 5 degrees for heading, and 1 m/s for vertical and horizontal velocities.

When the Hover-Stable-CO condition is satisfied, a Hover-Stable-EV event is sent to the DRC computer and an internal system flag is set to indicate that the system is in the autonomous mode and ready to accept new commands (SET PFC-Auto-Idle-AC). The *Mode Switch* automaton changes its state to Hovering-ST.

Extended State Machines. The HCSM framework has been further developed, resulting in Extended State Machines (ESMs) (Merz et al., 2006). The new framework adds several useful modeling features. The main changes include explicit modeling of task states, data flow, control and system flags, an event filtering mechanism and no explicit external events. It also includes a visual tool for designing and debugging state machines, facilitating the development of new and existing systems.

ESMs use three types of states: *simple states*, *superstates* (as in the HCSMs), and *task states*. Control and other functions are modeled explicitly in the task states. A schedule for function execution is provided by a scheduler included in the framework. The data used as input/output to the task state functions (data flow) is also explicitly modeled in the ESM formalism by *Data paths*. Asynchronous external events are modeled by a combination of *pulse event* and guard conditions. Thus only one internal event queue is used in the ESM. Additionally the ESM introduces an event filtering mechanism which limits the scope of internal events. The full details are available in Merz et al. (2006).

6.2 Interfacing between the Control Kernel and Higher Layers

6.2.1 The Platform Server

The *Platform Server* (Figure 11 on Page 22) encapsulates the functionality of the Control Kernel to provide a higher-level control and sensing interface for use by any of the functionalities in the reactive and deliberative layers. These functionalities interface to the Platform Server through two languages, the Flight Control Language (FCL) and Payload and Perception Control Language (PPCL) described in the following subsections. The former provides high-level parameterized commands which actuate the helicopter itself while the latter provides commands which actuate sensor payloads and call perception-based functionality. Many of these commands provide feedback to their callers in the form of events which are communicated through a shared memory resource. Each command is essentially associated with one or more HCSMs.

From one perspective, the commands associated with FCL and PPCL provide the primitive or basic actions provided by the UASTL RMAX system. Specification of more complex tasks used by different functionality in the reactive and deliberative layers use these basic actions as part of their definition. The intent is that most if not all use of the functionality in the control layer is accessed through the Platform Server interface.

This is a design decision that encapsulates the use of the control and sensor system in the UASTL RMAX through one common language interface. This makes the control kernel extensible in a straightforward manner. Any new control modes or addition of sensors are added to the system by implementing appropriate HCSMs to deal with real-time properties and temporal latencies near the hardware. The FCL and/or PPCL are then extended with an appropriate command interface.

Since the functionalities in the reactive and deliberative layers are only committed to soft real-time behavior while the HCSMs encapsulate real-time constraints, the Platform Server serves as an intermediary

between these different temporal latencies. The distributed nature of the HCSM framework allows different functionalities to run on different computers. Each of the on-board computers therefore executes its own HCSM interpreter and peer-to-peer RS232 serial lines are used to implement real-time communication channels between these. This allows the Platform Server to run on the DRC computer despite the fact that it must communicate with HCSMs associated with flight control modes and payload control which rely on timely execution and run in the real-time part of the system on the PFC and PPC computers.

Calling the Platform Server. The Platform Server runs on the DRC computer and provides a ROS-based interface to the commands available in the FCL and PPCL languages. When the server is asked to execute a specific command, it internally forwards the command through a lower-level programming interface based on the FCL and PPCL ("C-API" in Figure 11). This API in turn uses shared memory to forward the command to an HCSM executing in the real-time part of the DRC.

The HCSM on the DRC generates one or more events that are sent through the real-time communication channel to an HCSM on either the PFC or the PPC, depending on the command. The events generated on those computers, as well as any associated sensor data generated in the CK, are passed back to the Platform Server in the same manner.

FCL and PPCL commands are executed sequentially without an internal command queue for buffering commands. If a command has been accepted for execution, a success flag (FCL_OK or PPCL_OK) is returned. In case the system was already executing another command, a busy flag (FCL_BUSY) is returned. In case a new command has to be started immediately, the caller can first abort any currently running activity by sending a Cancel command.

Data Interface. An additional task of the Platform Server is to provide a Data Interface (DI) to the state of the UASTL RMAX system. The data periodically published through this interface contains the AV's state information (position, attitude, velocities), status information (such as the current engine RPM), payload information (such as the current position of the pan-tilt unit), and the results of perception-based functionalities (such as the color tracker). The Data Interface provides a common interface accessible by all deliberative and reactive functionalities. For example, the information obtained through the DI is used by the DyKnow system (Section 8) and by execution monitoring (Section 9.3).

6.2.2 FCL: Flight Control Language

The Flight Control Language (FCL) consists of a set of commands that are based on the control modes available in the Control Kernel and their parameterization. The reactive and deliberative services use the FCL commands for mission execution as it provides the interface to the available control functionalities in the CK.

The following are the most important commands currently available on the UASTL RMAX platform. The list is not exhaustive: Certain housekeeping commands, parameters, and return values are omitted here.

- **Take-off**: fcl_takeoff() takes off to a predefined altitude (using the take-off control mode). Returns:
  - FCL_BUSY: Another command is being executed.
  - FCL_NOT_ACCEPTED: Take-off can only be commanded when the AV is on the ground.
  - FCL_FINISHED.
- **Land**: fcl_land(heading) performs a vision-based landing (using the landing control mode). The parameter specifies the desired heading for landing. This heading should take into account the current position of the sun in order to avoid having the shadow of the AV fall on the landing pattern. Returns:
  - FCL_BUSY: Another command is being executed.
  - FCL_NOT_ACCEPTED: This command can only be issued if the AV is hovering.
  - FCL_FINISHED.

- **Yaw**: fcl_yaw(heading) changes the heading to a specific value (using the hovering control mode with a specified heading).
  Returns: The same values as for the fcl_land() command above.
- **Climb**: fcl_climb(altitude) climbs or descends to the given altitude (using the path following control mode with a predefined vertical path).
  Returns: The same values as for the fcl_land() command above.
- **Traj3d fly-to**: fcl_traj3d_flyto(longitude, latitude, altitude, velocity) flies to a position in a straight line (using the path following control mode with a single straight line segment). The parameters specify the world position to fly to and the desired cruising velocity.
  Returns: The same values as for the fcl_land() command above.
- **Traj3d**: fcl_traj3d(spline_description, velocity, end_velocity) flies between two waypoints following a spline specification (using the path following control mode). The parameters specify the beginning, ending, and direction vectors of the spline path, and the cruise / final segment velocities (Section 5.3).
  Returns: The same values as for the fcl_land() command above.
- **Track**: fcl_track() engages the object following control mode (uses the path following control mode, camera pan-tilt visual servoing control and vision-based color object tracking).
  Returns: The same values as for the fcl_land() command above.
- **Emergency brake**: fcl_emergency_brake() engages the emergency brake mode (path following and hovering control modes). This immediately aborts all flight control modes and engages the hovering control mode.
  Returns: FCL_OK.
- **Cancel**: fcl_cancel() cancels the execution of the current FCL command.
  Returns: FCL_OK.

For example, to scan an area in a given environment where the AV starts on the ground, the FCL commands used would include a *take-off*, a sequence of *fly-to* commands, and a *land* command.

6.2.3 PPCL: Payload and Perception Control Language

Similarly to the FCL, the Payload and Perception Control Language (PPCL) consists of a set of commands that are based on the modes available in the CK and their parameterization. The commands in this group relate to the use of payload control modes and perception functionalities.

The following are the most important PPCL commands currently available on the UASTL RMAX platform. The list is not exhaustive: Certain housekeeping commands, parameters, and return values are omitted here.

- **Request PTU, camera, laser range finder, perception functionality control**: Requests exclusive control and ownership for these functionalities. For example, the control over the pan-tilt unit and camera parameters cannot be granted during a vision-based landing. The landing control mode handles these parameters.
  - ppcl_request_ptu_control(): request control over the pan-tilt unit.
  - ppcl_request_cam_control(): request control over the camera parameters.
  - ppcl_request_ip_control(): request control over the image processing algorithms.
  - ppcl_request_laser_control(): request control over the laser range finder.

  Returns:
  - PPCL_CONTROL_DENIED
  - PPCL_OK

- **Release PTU, camera, laser range finder, perception functionality control**: Releases the control over these functionalities.
  - ppcl_release_ptu_control(): release control over the pan-tilt unit.
  - ppcl_release_cam_control(): release control over the camera parameters.
  - ppcl_release_ip_control(): release control over the image processing algorithms.
  - ppcl_release_laser_control(): release control over the laser range finder.

  Returns:
  - PPCL_OK
- **Pan-tilt unit control**: engages one of the pan-tilt control modes.
  - ppcl_look_at(longitude, latitude, altitude): world coordinates of a position at which the camera should be pointed.
  - ppcl_turn_to(pan_angle, tilt_angle): values of the pan-tilt unit axis to set.
  - ppcl_ptu_abort(): abort the current control mode.
- **Perception control**: engages one of the perception functionalities.
  - ppcl_ip_color_tracker(x, y): coordinates of a center of a object to track.
  - ppcl_ip_state_estimation(): engages the vision-based state estimation algorithm.
  - ppcl_ip_abort(): abort the current command.
- **Camera parameters control**: sets the required camera parameters.
  - ppcl_zoom(value): the value of the camera zoom from 0 (full zoom out) to 1 (full zoom in).
  - ppcl_exposure(auto): specifies whether automatic or manual exposure should be used.
  - ppcl_iris(value): the value of the camera iris to set.
  - ppcl_shutter(value): the value of the camera shutter to set.
- **Laser parameters control**: sets the required laser range finder parameters.
  - ppcl_laser_angular_resolution(value): the angular resolution of the laser sensor (1, 0.5, 0.25 deg).
  - ppcl_laser_distance_resolution(value): the distance resolution of the laser sensor (1 mm, 1 cm).
  - ppcl_laser_rotation_speed(value): the speed and direction of the laser rotation mechanism.
  - ppcl_laser_rotation_angle(value): the angle of the laser rotation mechanism to set.
- **Laser control**: engages one of the laser perception functionalities.
  - ppcl_laser_start_data_collection(): starts collection of the laser range finder data.
  - ppcl_laser_stop_data_collection(): stops collection of the laser range finder data.
- **Pan-Tilt activity info**: returns which pan-tilt control mode is currently active.
  - ppcl_ptu_activity_info().

  Returns:
  - PPCL_LOOK_AT_POINT
  - PPCL_TURNTO
  - PPCL_IDLE
- **Camera parameters**: returns parameters of a specific camera.
  - ppcl_get_camera_info(index): index of the camera (color or thermal)

  Returns: Intrinsic camera parameters.

- **Pan-tilt parameters**: returns parameters of the pan-tilt mechanism.
  - ppcl_min_pan_value(), ppcl_max_pan_value(), ppcl_min_tilt_value(), ppcl_max_tilt_value():

  Returns: Ranges of pan-tilt unit angles which can be set.

In the case of scanning an area in a given environment, the required PPCL command sequence would include *request control: pan-tilt*, *request control: camera*, *pan-tilt control: tilt 90 degrees*, *camera control: set specific zoom*, *release control: pan-tilt*, and *release control: camera*.

6.2.4 DI: Data interface

Along with the FCL and PPCL languages, the control kernel also provides information about the state of the UASTL RMAX platform. This data is made available to the higher layers periodically at the rate of 25 Hz. It is currently distributed in the system in the form of ROS topics. The two most commonly used data collections provided by this interface are:

- **Helicopter state**:
  - altitude, latitude, longitude: current position.
  - velocity_north, velocity_east, velocity_up: current velocity in the given direction.
  - pitch, roll, heading: current pitch angle, roll angle and heading.
  - rpm: engine revolutions per minute.
  - on_ground_flag: true if the AV is on the ground.
  - auto_flag: true if the AV is in autonomous flight mode.
  - drc_minus_pfc_time, ppc_minus_pfc_time: time differences between computers.
  - power_status: indicates which on-board devices are switched on.
  - pfc_time: data timestamp.
- **Camera state**:
  - id: id of the camera (color or thermal).
  - pan, tilt: current position of the PTU's pan and tilt axis, respectively.
  - zoom: current zoom factor of the camera.
  - ipc_time: timestamp.

6.3 Task Specification Trees

The concept of a *task* has many different interpretations in the literature and has been instantiated in many different ways in architectures proposed for mobile robots. One way to specify a particular type of task that has strict real-time requirements is using HCSMs. This type of task is often called a *behavior* in the robotics literature (Arkin, 1998; Konolige et al., 1997) and different ways are proposed for combining such behaviors. At the other end of the spectrum, a task is often interpreted as an action or combination of actions (composite action) that are combined using standard control structures such as sequentialization, concurrency, conditional branching, etc. In fact, there have been several proposals to simply implement robot behaviors and tasks using a subset of a conventional programming language such as C (Konolige, 1997). Firby's Reactive Action Packages (Firby, 1987) is one example where tasks are specified using a specialized language which combine actions with control. RAPs are often interpreted as *compiled plans*. Automated planners themselves combine actions specified as plan operators. The output of any planner can be interpreted as a composite action or task. The transition from plans to executable tasks is an essential part of any intelligent system.

The choice and construction of a task specification language which covers much of the spectrum of interpretations above has to be chosen with generality in mind in order to be used as a common interface to many diverse functionalities, but at the same time, the actual tasks have to be efficiently executable. The choice is further complicated by the fact that when specifying collaborative mission scenarios, tasks are often shared and must be distributable. An additional factor is that in the context of heterogeneous robotic systems, the specification of tasks should include context. For example, the **fly-to** action of a fixed wing platform and the **fly-to** action of a rotor based platform have much in common but also much that is different.

Task Specification Trees (TSTs) have been proposed as a general task specification language which meets many of the requirements stated above. Each node in such a tree is a specification of an elementary action such as a ROS call to an FCL command, a control structure such as sequential, concurrent, branching or loop execution, or a goal for which a plan must be generated and executed (Section 9.2). In the latter case, the tree is *dynamically expandable* as a plan is generated. Goal nodes are therefore allowed to add children to themselves, after which they act as sequence or concurrency nodes. TSTs also allow for the statement of context in the form of constraints which can then be checked for consistency with a constraint solver. Additionally, these constraints are often temporal and spatial in nature and implicitly combine both a reasoning and procedural execution mechanism for tasks.

A TST is in itself purely declarative, defining what should be achieved and providing parameters and constraints for tasks. For example, a sequence node declaratively specifies that its children should be sequentially executed while a **fly-to** node would specify that an aircraft should fly to a specific location, usually with associated parameters such as the intended speed and altitude, possibly constrained by context.

At the same time there is a close integration with execution aspects of actions and action composition constructs through the coupling of *executors* to specific types of TST nodes. Each executor is an executable procedure specifying *how* a task of the given type should be performed. In many cases, the interface provided by the Platform Server is used to issue commands in the Flight Command Language and Payload and Perception Control Language (Section 6.2). The implementation of the executor can be platform-specific, allowing the exact mode of execution for a particular task to be platform-dependent as is often necessary when heterogeneous platforms are used. Thus, the use of executors provides a clear separation between task specifications and platform-specific execution details.

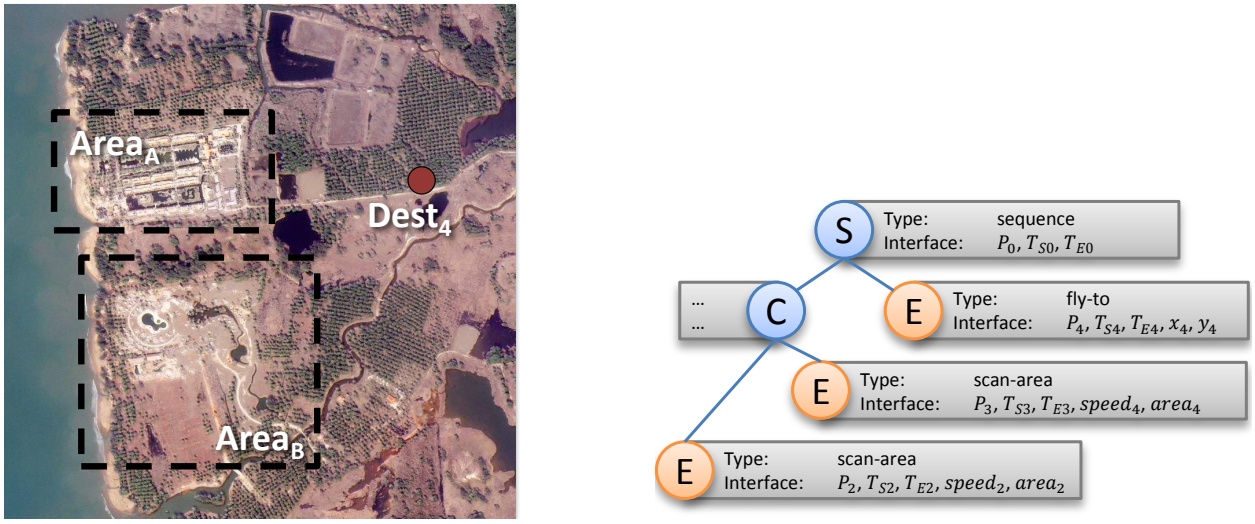
In the case of the UASTL RMAX system, nodes for actions such as take-off, vision-based landing, hovering or path following are generally associated with executors calling the Platform Server to initiate the corresponding autonomous flight modes. Through this interface TST nodes can also control the payload and sensors of the UAS platform and receive associated information including image processing results. Note that executors are required even for *control nodes* or *structural nodes*, though such executors may be identical across platforms. For example, a sequential node requires an executor that procedurally ensures its children are executed in sequential order, possibly with additional timing constraints.

TSTs have been applied and tested successfully in a number of deployed UAS systems (Doherty et al., 2013, 2010; Landén et al., 2010).

Example 6.1 *Consider a small scenario similar to the first goal in the example described in the introduction. The mission is that two AVs should concurrently scan the areas* Area_A *and* Area_B*, after which the first AV should fly to* Dest_4 *(Figure 15(a)). The corresponding TST (Figure 15(b)) uses three* elementary action nodes *(marked E), corresponding to two elementary actions of type* ***scan-area*** *and one of type* ***fly-to****. Furthermore, it requires a* concurrent node *(marked C) specifying that the* ***scan-area*** *actions can be performed concurrently, as well as a* sequential node *(marked S). Further explanations will be given below.*

6.3.1 Task Specification Tree Structure

Task Specification Trees are implemented as a distributed data structure where nodes can reside on any platform, linked to their parents and children on the same or another platform. Each platform involved in



(a) Map of the mission area.

(b) Example Task Specification Tree.

Figure 15: Example mission.

a mission has a TST manager accessible through ROS-based service calls, allowing communication and synchronization between nodes when necessary.

As illustrated in Figure 15(b), each *type* of node is associated with a set of *node parameters*. Together, these parameters are called the *node interface*. The interface always contains a platform (agent) assignment parameter, usually denoted by P_i, which identifies the agent responsible for executing the task. There are also always two parameters for the start and end times of the task, usually denoted by T_{Si} and T_{Ei}, respectively. Tasks often have additional type-specific parameters, such as a speed parameter and an area parameter for a **scan-area** action node.

When a concrete TST is specified, some node parameters may be given actual values through arguments. In the example, one may specify that the areas to scan are Area_A and Area_B (not shown in the figure). Other parameters may be left open. Concrete values may then be chosen by the executing agent, subject to a set of *constraints* associated with each node. These constraints can be used to constrain permitted values for the parameters of the same node as well as all parameters of ancestor nodes in the tree. By constraining temporal parameters, one can also express precedence relations and organizational relations between the nodes in the TST that are not implicitly captured by the use of specific control nodes. Together the constraints form a constraint network where the node parameters function as constraint variables.

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

6.3.2 Task Specification Tree Language

For convenience, a text-based language for representing Task Specification Trees has also been defined (Figure 16). The `TST` construct corresponds to a specific parameterized node and introduces the main recursive pattern. All such nodes must be explicitly named in order to allow name-based references. The parameters provided after the name specify the node interface which can be accessed from the outside. These can be constrained relative to each other using constraints in a **where** clause.

The formal semantics of a TST is specified through a translation from this language into composite actions in Temporal Action Logic (Doherty and Kvarnström, 2008; Doherty et al., 2012). This provides a means of formally verifying the behavior of tasks and is also useful as a debugging tool.

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

Figure 16: Task Specification Tree language

Example 6.2 *Consider again the TST depicted in Figure 15(b). This TST contains two composite actions, sequence (here named τ_0) and concurrent (τ_1), and two elementary actions, scan (τ_2, τ_3) and flyto (τ_4). nodes in this TST have the task names τ_0 to τ_4 associated with them.*

$\tau_0(T_{S_0},T_{E_0})$ = **with** $T_{S_1},T_{E_1},T_{S_4},T_{E_4}$ **sequence**
  $\tau_1(T_{S_1},T_{E_1})$ = **with** $T_{S_2},T_{E_2},T_{S_3},T_{E_3}$ **concurrent**
    $\tau_2(T_{S_2},T_{E_2})$ = **scan-area**$(T_{S_2},T_{E_2},Speed_2,Area_A)$;
    $\tau_3(T_{S_3},T_{E_3})$ = **scan-area**$(T_{S_3},T_{E_3},Speed_3,Area_B)$
  where $cons_{\tau_1}$;
  $\tau_4(T_{S_4},T_{E_4})$ = **fly-to**$(T_{S_4},T_{E_4},Speed_4,Dest_4)$
where $cons_{\tau_0}$

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

The use of TSTs which call path planners are described in the next section. Additional features of TSTs and their relation to temporal action logic and high level mission specifications are described in Section 9.4.

7 The Navigation Subsystem

Many of the missions where AVs are deployed require sophisticated path planning capability. One might need to fly directly to a particular waypoint, or in more complex scenarios, one might require the generation of a segmented path which is guaranteed to be collision free. Many of the TSTs used in actual missions are required to call a motion planner which then outputs such segmented paths. The TST is then expanded with this output and executed. The navigation subsystem is responsible for this complex combination of processes which begin with a call to a motion planner at the deliberative layer. The output of the motion planner and its execution involves numerous calls to the Platform Server through use of FCL commands which in turn initiate execution of appropriate HCSMs. These in turn use the continuous control modes implemented in the Control Kernel.

This section provides a detailed description of the processes involved in the generation and execution of motion plans on the UASTL RMAX system. Additionally, it describes two path planning algorithms integrated in the HDRC3 architecture based on extensions (Pettersson, 2006; Pettersson and Doherty, 2006)

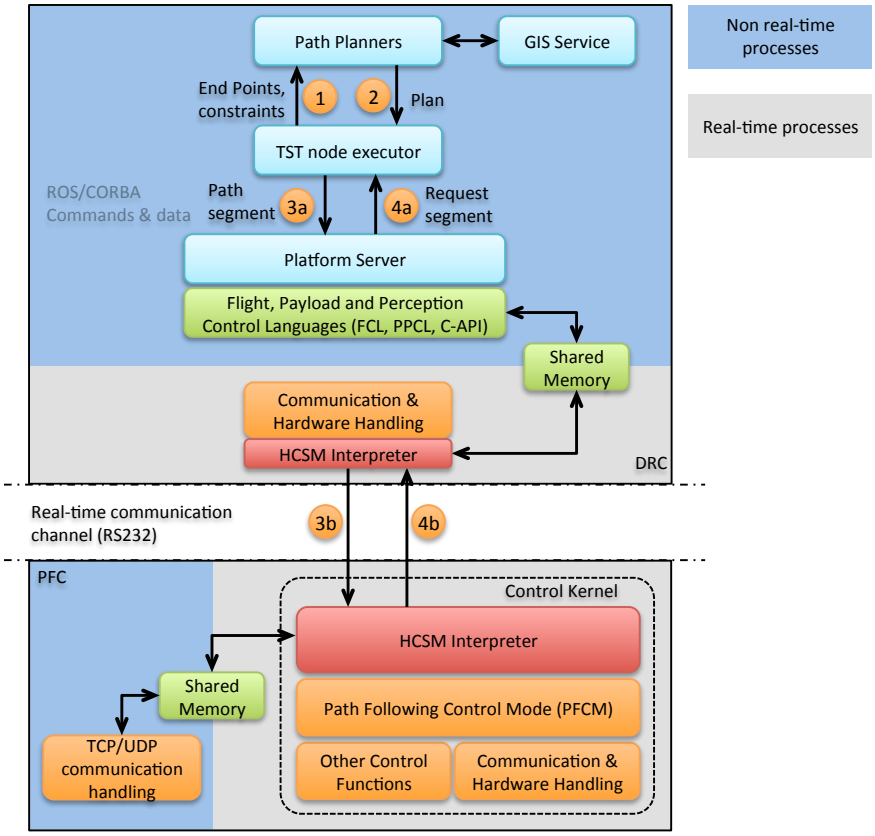


Figure 17: Navigation subsystem and main software components

to two sample-based planning methods: Probabilistic RoadMaps (PRM (Kavraki et al., 1996)) and Rapidly-Exploring Random Trees (RRT (Kuffner and LaValle, 2000)).

Navigation Scheme. The standard navigation scheme in the HDRC3 architecture, assuming static operational environments, is depicted in Figure 17. The task of flying is represented through an action node in a TST. For example, the task of flying to a specific waypoint can be represented declaratively as an elementary action node of type **fly-to**. The executor for nodes of this type calls a path planner (step 1 in the figure) that takes a map of static and dynamic obstacles together with the initial position, goal position, desired velocity and possibly a set of additional constraints. The path planner then generates a segmented path (see Section 7.1) which is represented as a sequence of cubic polynomial curves. The generated path is collision-free relative to the world model provided by a *Geographic Information System* (GIS Service). Each segment of the path is defined by start and end points, start and end directions, target velocity and end velocity. If successful, this segmented path is returned to the TST node executor (step 2).

Once a segmented path is returned, the TST node executor requires a suitable interface to the low-level Path Following Control Mode (PFCM). The PFCM implements continuous control laws which place certain real-time requirements on the way they are used and the mode execution is therefore coordinated by HCSM state machines. The TST node executor therefore sends the first segment of the path via the Platform Server (step 3) and waits for a *Request Segment* event to be returned. This event is generated by the HCSM responsible for the path execution as soon as the PFCM controller receives a path segment as input. Details will be discussed in Section 7.2.

When a *Request Segment* event arrives (step 4) the TST node executor sends the description of the next segment to the HCSM that coordinates path execution at the control level. This procedure is repeated (steps 3–4) until the last segment is executed. However, because the high-level system is not implemented in hard real-time it may happen that the next segment does not arrive at the Control Kernel on time. In this case, the controller has a timeout limit after which it goes into safety braking mode in order to stop and hover at

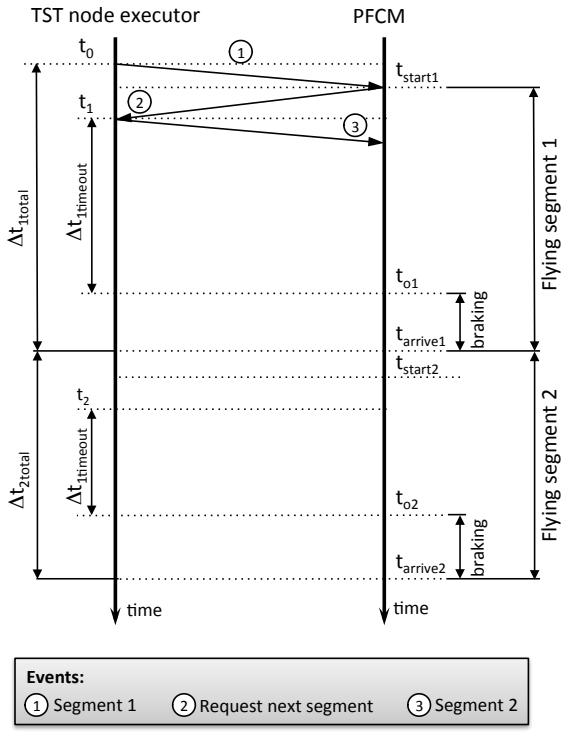


Figure 18: Execution time-line for a path consisting of 2 segments.

the end of the current segment. The timeout is determined by a velocity profile dynamically generated for the path segment together with the current position and current velocity.

Figure 18 depicts a timeline plot of the execution of a 2-segment trajectory. At time t_0, a TST node executor sends the first segment of the path to the PFCM controller and waits for a *Request segment* event which arrives immediately (t_1) after the helicopter starts to fly (t_{start1}). Typical time values for receiving a *Request segment* event ($t_1 - t_0$) are well below 200 ms. Time t_{o1} is the timeout for the first segment which means that the TST node executor has a $\Delta_{t_1 timeout}$ time window to send the next segment to the PFCM controller before it initiates a safety braking procedure. If the segment is sent after t_{o1}, the helicopter will start braking. In practice, the $\Delta_{t_1 timeout}$ time window is large enough to replan the path using the standard path planner (Section 7.1). The updated segments are then sent to the PFCM controller transparently.

The path execution mechanism described here allows for dynamic replacement of path segments if necessary. This is an important part of the architecture due to the fact that it is assumed that higher-level deliberative components are continually planning, executing and monitoring missions which are likely to change due to contingencies. This architectural solution supports a certain type of any-time behavior in the system which takes into account resource constraints when reconfiguring plans (Wzorek, 2011).

7.1 Path and Motion Planning

Path planning and motion planning algorithms deal with the problem of generating collision-free paths for a robot in order to navigate or move in an physical space, called the *workspace* $\mathcal{W}$. The workspace is most often modeled as $\mathcal{R}^3$ but can be restricted to $\mathcal{R}^2$ for robots navigating in a single plane. This type of representation is particularly well-suited for collision checking since the robot and the obstacles are represented in the same space. However in many practical applications the workspace is not sufficient to describe the planning problem and a more expressive representation is required.

The *configuration space* (*C* or *C-space*) is defined as a vector space or manifold of configurations q, where a configuration is a set of parameters that uniquely defines the location of all points of the robot in the workspace $\mathcal{W}$. For a rigid-body robot such as an AV platform this would include not only its position

but also its orientation. Additionally, not all robot configurations are attainable due to obstacle constraints. The *free space* denoted by C_{free} is a subset of the C-space that is free from collisions with obstacles.

When dealing with robotic systems in motion, the configuration of the robot is insufficient to describe the problem: The dynamic state of a robot (its velocity) also has to be accounted for. The *state space* representation extends the configuration space by adding first-order derivatives $\dot{q}$ of the robot configuration q. Thus, for a robot configuration $q = (q_0, \ldots, q_n)$, the state x is defined by $x = \langle q, \dot{q} \rangle$ where $\dot{q} = (\dot{q}_0, \ldots, \dot{q}_n)^T$.

Constraints. In addition to the requirement of avoiding collisions, plans must also satisfy *kinematic* and *dynamic* constraints. Kinematic constraints include only first-order derivatives of the configuration parameters, while second-order derivatives such as acceleration are allowed in the dynamic constraints. The algorithms presented below handle the kinematic and dynamic constraints of the UASTL RMAX platforms.

Both of these types of constraints belong to a class of non-holonomic constraints (also called motion constraints) and are common for many types of robots. A robot is non-holonomic if it has fewer controllable degrees of freedom than total degrees of freedom. A car is non-holonomic since it can only drive forwards or backwards, not sideways. So is a helicopter: Though it can move freely in any direction, its freedom of movement depends on its speed. When a helicopter is hovering or flying slowly it could be considered to be holonomic, but this would constrain its usage.

Path Planning. The path planning problem is defined as finding a path in C_{free} that connects the start (q_0) and the goal (q_g) configuration. For high-dimensional configuration spaces this problem is intractable in general which has led to the development of sample-based methods such as PRMs (Kavraki et al., 1996) and RRTs (Kuffner and LaValle, 2000). These use an approximation of the C_{free} continuous space (in configuration space or state-space) in order to deal with the complexity of high-dimensional problems. The discrete representation of the original continuous space (typically represented in the form of a graph) sacrifices strict completeness for a weaker definition such as *resolution completeness* or *probabilistic completeness* (LaValle, 2004).

7.1.1 Probabilistic Roadmaps and Rapidly Exploring Random Trees

The standard probabilistic roadmap (PRM) algorithm (Kavraki et al., 1996) works in two phases, one offline and the other on-line. In the offline phase, a discrete roadmap representing the free configuration space is generated using a 3D world model. First, it randomly generates a number of configurations and checks for collisions with the world model. A local path planner is then used to connect collision-free configurations taking into account kinematic and dynamic constraints of the helicopter. Paths between two configurations are also checked for collisions. This results in a roadmap approximating the configuration free space. In the on-line or querying phase, start and goal configurations are provided and an attempt is made to connect each configuration to the previously generated roadmap using a local path planner. Finally, a graph search algorithm is used to find a path from the start configuration to the goal configuration in the augmented roadmap.

Figure 19 provides a schema of the PRM path planner that is used. The planner uses an Oriented Bounding Box Trees (OBBTrees) algorithm (Gottschalk et al., 1996) for collision checking and an A* algorithm for graph search. Here one can optimize for various criteria such as shortest path, minimal fuel usage, etc.

The mean planning time in the current implementation for a selected set of flight test environments is below 1000 ms and the use of runtime constraints (discussed below) does not noticeably influence the mean. See Pettersson (2006); Pettersson and Doherty (2006) for a detailed description of the modified PRM planner.

Rapidly exploring random trees (RRT (Kuffner and LaValle, 2000)) provide an efficient motion planning algorithm that constructs a roadmap online rather than offline (Figure 19). The RRT algorithm generates two trees rooted in the start and goal configurations by exploring the configuration space randomly in both

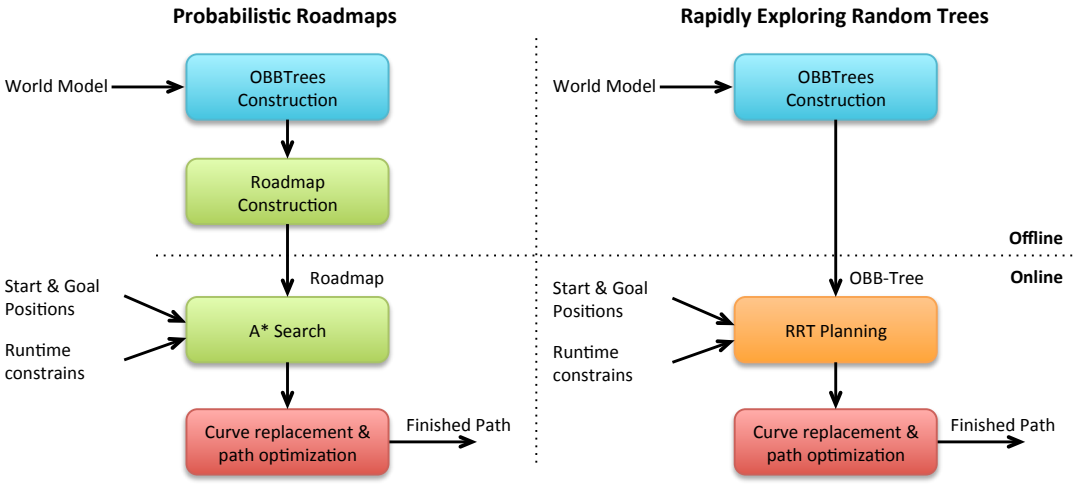


Figure 19: PRM and RRT path plan generation.

directions. While the trees are being generated, an attempt is made at specific intervals to connect them to create one roadmap. After the roadmap is created, the remaining steps in the algorithm are the same as with PRMs. The mean planning time with RRT is also below 1000 ms, but the success rate is considerably lower and the generated plans may sometimes cause anomalous detours (Pettersson, 2006). The UASTL RMAX system uses both the PRM and RRT planners individually and in combination.

7.1.2 Path Planner Extensions

The standard PRM and RRT algorithms are formulated for fully controllable systems only. This assumption is true for a helicopter flying at low speed with the capability to stop and hover at each waypoint. However, when the speed is increased the helicopter is no longer able to negotiate turns of a smaller radius. This in turn imposes demands on the planner similar to non-holonomic constraints for car-like robots.

The most straightforward way of handling dynamic constraints in the PRM and RRT algorithms is to complement the configurations with their derivatives and record the complete state at each node in the graph. This enables the local path planner to adapt the path between two nodes to their associated derivatives, which is necessary to respect the dynamic constraints at boundary points between adjacent edges in the solution path. The drawback of this approach is that the dimensionality of the space in which the roadmap/tree is situated is doubled thus increasing the complexity of the problem. An alternative approach to non-holonomic planning is to postpone the non-holonomic constraints to the runtime phase. The following extensions have therefore been made with respect to the standard version of the PRM and RRT algorithms.

Multi-level roadmap/tree planning. Inspired by a multi-level planner proposed by Sekhavat et al. (1996), new planners for AV applications have been developed (Pettersson, 2006; Pettersson and Doherty, 2006). In this approach, linear paths are first used to connect configurations in the graph/tree and at a later stage these are replaced with cubic curves when possible (Figure 20). These are required for smooth high speed flight. If it is not possible to replace a linear path segment with a cubic curve then the helicopter has to slow down and switch to hovering mode at the connecting waypoint before continuing. This rarely happens in practice.

The random sampling nature of these planners and the limited density of the sampled nodes make the PRM and RRT algorithms produce paths that are often jagged and irregular with occasional detours. In order to improve the quality of the paths, a smoothing step is usually added. For the implemented path planners the following smoothing steps are performed:

- Node Alignment: For each node n along the path, two attempts are made to move it to a point

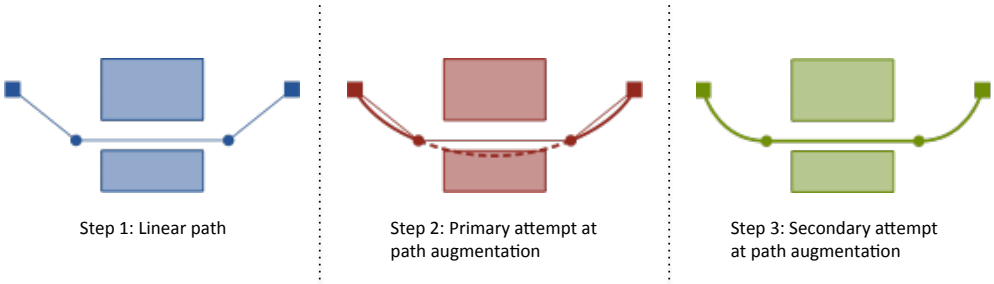


Figure 20: Transformation from linear to cubic path segments for smooth flight.

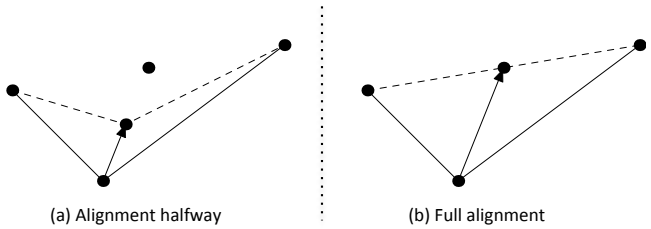


Figure 21: Alignment of nodes for improved path quality.

that straightens out the path (Figure 21). The point m in the middle between the two neighbors is located. First, an attempt is made to move n halfway to m (Figure 21a), if this is possible given known obstacles. Then an attempt is made to move it all the way to m (Figure 21b).

- Node Elimination: For each node along the path, an attempt is made to eliminate it by connecting the two adjacent nodes directly. If the connection satisfies all constraints, the middle node is eliminated.

The curve replacement step described above is performed between the alignment and the elimination step.

Runtime constraint handling. Some constraints are not available during roadmap or tree construction but are added at runtime, when a final plan is requested. The motion planner currently handles the following types of runtime constraints during the A* search phase:

- Maximum and minimum altitude – verified through an intersection test between the configuration or curve and a horizontal plane.
- Forbidden regions (no-fly zones) – regions created by a set of horizontal polygons covering an area which the AV must not enter. The satisfaction test for configurations involves checking if the AV's position, projected in the X/Y plane, is within the polygon. The test for curves includes checking if the curve intersects any of the planes bounding the polygon in the 3D-space.
- Limits on the ascent and descent rate.

Adding constraints to a path planning problem may break the connectivity of the PRM roadmap. In this case a combination of the PRM and RRT algorithms is used to mitigate the problem. An optional reconnection attempt using the RRT planner is made during the query phase of the PRM planner in order to reconnect the broken roadmap. Both planners (PRM and RRT) and their combination with the presented extensions are accessible to the deliberative and reactive services in the HDRC3 architecture.

7.2 HCSMs for Path Execution

The use case described in this section assumes the UASTL RMAX system is already in autonomous flight mode and the default hovering function is active (for example after executing the use case presented in

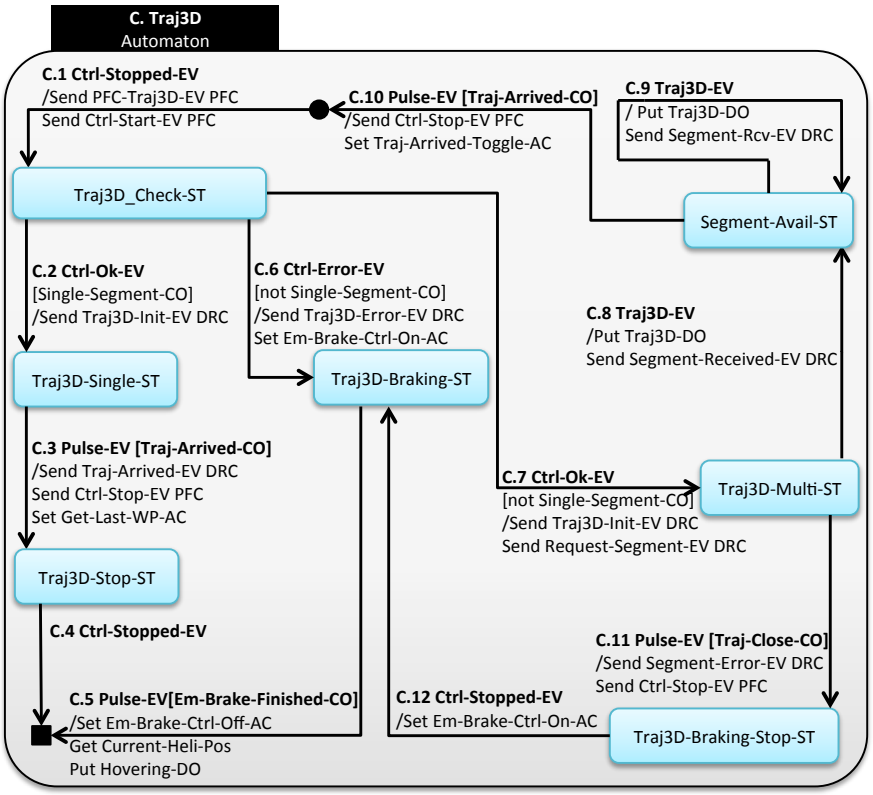


Figure 22: The *Traj3D* automaton.

Section 6.1). The description focuses on the execution of the path at the lowest control level running on the PFC computer after a TST node executor has received a path plan (a segmented cubic polynomial curve) from the path planner as described at the beginning of previous section. For better understanding of the interactions between the HCSM automata the focus will be on the execution of one path segment.

As previously stated, the TST node executor sends the first segment (step 3 in Figure 17, Traj3D-EV) of the trajectory via the *Platform Server* and waits for a Request-Segment-EV event that is generated by the controller. At the control level, the path is executed using the Path Following Control Mode (PFCM, described in Section 5.2). When a Request-Segment-EV event arrives (step 4) the TST node executor sends the next segment. This procedure (steps 3–4) is repeated until the last segment is sent. However, because the high-level system is not implemented in hard real-time it may happen that the next segment does not arrive at the Control Kernel on time. In this case, the controller has a timeout limit after which it goes into safety braking mode in order to stop and hover at the end of the current segment.

As described in Section 6.1 the implementation of the PFCM function follows a predefined design template. The function itself is executed by setting an appropriate internal flag using a SET action (SET Traj3D-Mode-AC, B.3) by the *Control Mode* automaton. *Mode Switch* makes sure the default hovering function is properly terminated before the PFCM function can be activated. Additionally, when path execution is finished it engages the default hovering function, in a manner similar to the example shown previously.

The role of the *Traj3D* automaton (Figure 22) is to coordinate an exchange of events with other services (such as TST node executors) and to ensure that the appropriate segment data is available to the PFCM control function when needed. An example time-line for a path execution showing the interaction between the three automata is presented in Figure 23. The example focuses on the execution of a single segment.

The *Mode Switch* automaton starts in the Hovering-ST state. After receiving a Traj3D-EV event from the DRC computer, the data describing a segment for execution is saved in the memory slot used by the PFCM function (Traj3D-DO). Additionally, the Ctrl-Stop-EV event (A.6) is sent to the *Control Mode* automaton in order to stop the execution of the default hovering function. At this time, *Mode Switch* transitions to the

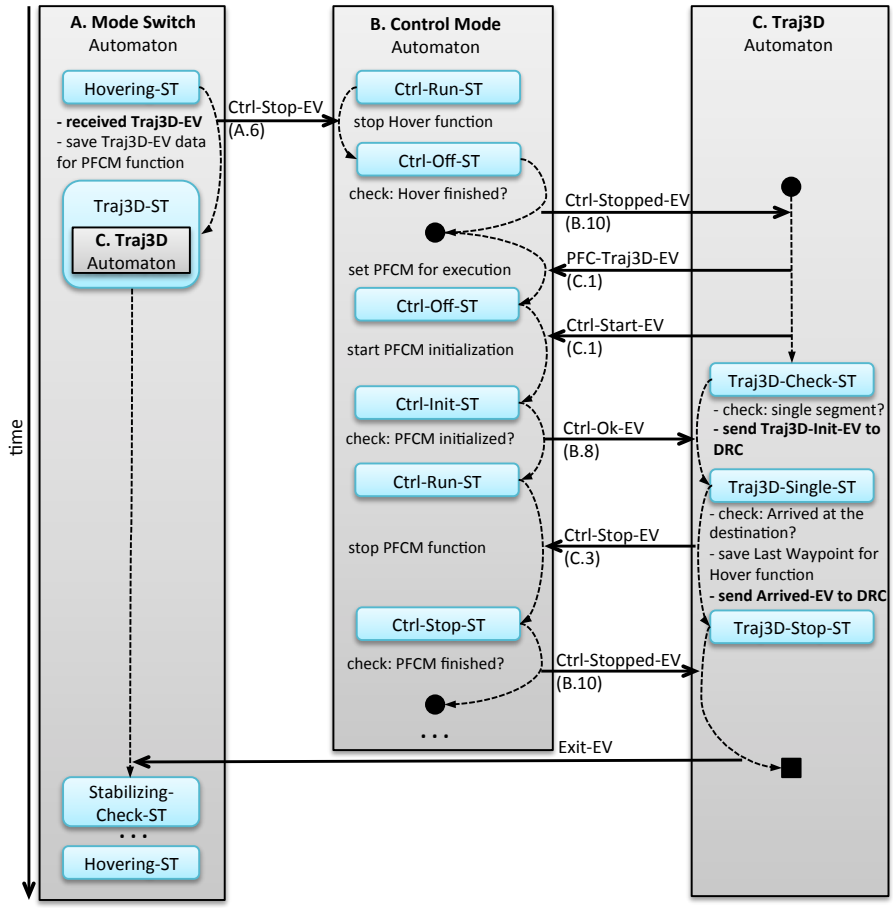


Figure 23: Execution trace: The interaction between *Control Mode*, *Mode Switch* and *Traj3D*.

Traj3D-ST state (Figure 22), thereby starting the *Traj3D* automaton in its *init* state. *Mode Switch* remains in Traj3D-ST until the PFCM execution is finished and then switches to the default hovering mode.

When *Traj3D* starts, it waits for confirmation from *Control Mode* that the hovering function has been terminated (Ctrl-Stopped-EV, B.10) and sends two events back to initialize and start the execution of the PFCM control function (PFC-Traj3D-EV (C.1) and Ctrl-Start-EV (C.1)). It transitions to the Traj3D-Check-ST state waiting for confirmation that the PFCM function has been initialized (Ctrl-Ok-EV, B.8). When the event arrives the HCSM checks whether a single segment has been received (Single-Segment-CO, C.3). A single segment is defined by the path parameters with the end velocity for the segment set to zero. If the condition is satisfied, the event informing the DRC computer that the segment has been accepted for execution is sent.

At this time the PFCM starts the execution of the segment. The *Traj3D* automaton is in the Traj3D-Single-ST state and remains there until the Traj-Arrived-CO condition is not satisfied. On the UASTL RMAX the segment has been successfully executed when the distance to the final waypoint is less than 3 meters. At that point, the last waypoint of the path is saved for the hovering function and the Traj-Arrived-EV event is sent to the DRC computer informing it that the path execution is finished. Additionally, the *Traj3D* automaton stops the execution of the PFCM function by sending the Ctrl-Stop-EV event (C.3) to the *Control Mode* state machine. When the termination of the execution is confirmed by receiving the Ctrl-Stopped-EV event (B.10) the *Traj3D* automaton transitions to its exit state and the *Mode Switch* state machine takes care of engaging the default hover mode, in a manner similar to the example described previously (Section 6.1.1).

8 DyKnow: Stream-Based Reasoning Middleware

For autonomous unmanned aerial systems to successfully perform complex missions, a great deal of embedded reasoning is required. For this reasoning to be grounded in the environment, it must be firmly based on information gathered through available sensors. However, there is a wide gap in abstraction levels between the noisy numerical data directly generated by most sensors and the crisp symbolic information that many reasoning functionalities assume to be available. This is commonly called the *sense-reasoning gap*.

Bridging this gap is a prerequisite for deliberative reasoning functionalities such as planning, execution monitoring, and diagnosis to be able to reason about the current development of dynamic and incompletely known environments using representations grounded through sensing. For example, when monitoring the execution of a plan, it is necessary to continually collect information from the environment to reason about whether the plan has the intended effects as specified in a symbolic high-level description.

Creating a suitable bridge is a challenging problem. It requires constructing representations of information incrementally extracted from the environment. This information must continuously be processed to generate information at increasing levels of abstraction while maintaining the necessary correlation between the generated information and the environment itself. The construction typically requires a combination of a wide variety of methods, including standard functionalities such as signal and image processing, state estimation, and information fusion.

These and other forms of reasoning about information and knowledge have traditionally taken place in tightly coupled architectures on single computers. The current trend towards more heterogeneous, loosely coupled, and distributed systems necessitates new methods for connecting sensors, databases, components responsible for fusing and refining information, and components that reason about the system and the environment. This trend makes it less practical to statically predefine exactly how the information processing should be configured. Instead it is necessary to configure the way in which information and knowledge is processed and reasoned about in a context-dependent manner relative to high-level goals while globally optimizing the use of resources and the quality of the results.

To address these issues, the stream-based reasoning middleware DyKnow (Heintz, 2009; Heintz and Doherty, 2004; Heintz et al., 2010) has been developed. This is a central part of the HDRC3 architecture as shown in Figure 1.

8.1 DyKnow

The main purpose of DyKnow is to provide generic and well-structured software support for the processes involved in generating state, object, and event abstractions about the environments of complex systems. The generation is done at many levels of abstraction beginning with low level quantitative sensor data and resulting in qualitative data structures which are grounded in the world and can be interpreted as knowledge by the system. To produce these structures DyKnow supports operations on streams at many different levels of abstraction. For the result to be useful, the processing must be done in a timely manner so that a UAS can react in time to changes in the environment. The resulting structures are used by various functionalities in the HDRC3 architecture for situation awareness and assessment (Heintz et al., 2007), planning to achieve mission goals (Section 9.2), and monitoring (Section 9.3). DyKnow provides a declarative language for specifying the structures needed by the different subsystems. Based on this specification it creates representations of the external world and the internal state of an AV based on observations and a priori knowledge, such as facts stored in databases.

DyKnow helps organize the many levels of information and knowledge processing in a distributed robotic system as a coherent network of processes connected by streams. The streams contain time-stamped information and may be viewed as representations of time-series data which may start as continuous streams from sensors or sequences of queries to databases. Eventually, they will contribute to more refined, composite, knowledge structures. Knowledge producing processes combine streams by applying functions,

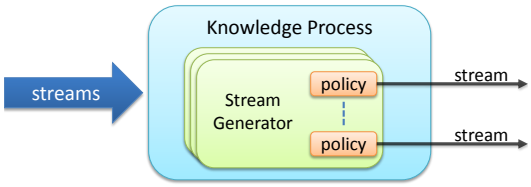


Figure 24: A prototypical knowledge process.

synchronization, filtering, aggregation and approximation as they move to higher levels of abstraction. In this sense, DyKnow supports conventional data fusion processes, but also less conventional qualitative processing techniques common in the area of artificial intelligence. In Heintz and Doherty (2006) it is argued that DyKnow supports all the functional abstraction levels in the JDL Data Fusion Model (White, 1988).

A knowledge process has different quality of service properties such as maximum delay, trade-off between quality and delay, how to calculate missing values, and so on, which together define the semantics of the knowledge derived 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 data may be processed differently by different parts of the architecture by tailoring the knowledge processes relative to the needs and constraints associated with the tasks at hand. This allows DyKnow to support easy integration of existing sensors, databases, reasoning engines and other knowledge producing services.

A knowledge processing application in DyKnow consists of a set of *knowledge processes* (Figure 24) connected by *streams* satisfying *policies*. A policy is a declarative specification of the desired properties of a stream. Each knowledge process is an instantiation of a *source* or a *computational unit* providing *stream generators* that produce streams. A source makes external information, such as the data provided by the Data Interface in the Platform Server (Section 6.2.1), available in the form of streams while a computational unit refines and processes streams.

8.2 Streams and Policies

Knowledge processing for a physical agent is fundamentally incremental in nature. Each part and functionality in the system, from sensing to deliberation, needs to receive relevant information about the environment with minimal delay and send processed information to interested parties as quickly as possible. Rather than using polling, explicit requests, or similar techniques, the strategy is to model and implement the required flow of data, information, and knowledge in terms of *streams* while computations are modeled as active and sustained *knowledge processes* ranging in complexity from simple adaptation of raw sensor data to complex reactive and deliberative processes.

Streams lend themselves easily to a *publish/subscribe* architecture. Information generated by a knowledge process is published using one or more *stream generators* each of which has a (possibly structured) *label* serving as an identifier within a knowledge processing application. Knowledge processes interested in a particular stream of information can subscribe to it using the label of the associated stream generator which creates a new stream without the need for explicit knowledge of which process hosts the generator. Information produced by a process is immediately provided to the stream generator, which asynchronously delivers it to all subscribers, leaving the knowledge process free to continue its work. Using an asynchronous publish/subscribe pattern of communication decouples knowledge processes in time, space, and synchronization, providing a solid foundation for distributed knowledge processing applications.

Each stream is associated with a declarative *policy*, a set of requirements on its contents. Such requirements may include the fact that elements must arrive ordered by valid time, that each value must constitute a significant change relative to the previous value, that updates should be sent with a specific sample fre-

quency, or that there is a maximum permitted delay. Policies can also give advice on how these requirements should be satisfied, for example by indicating how to handle missing or excessively delayed values.

8.3 Objects, Features and Fluent Streams

For modelling purposes, the environment of an AV is viewed as consisting of physical and non-physical *objects*, *properties* associated with these objects, and *relations* between these objects. The properties and relations associated with objects are called *features*. Due to the potentially dynamic nature of a feature, that is, its ability to change values through time, a total function from time to value called a *fluent* is associated with each feature. It is this fluent, representing the value over time of a feature, which is being modeled. Example objects are *the AV*, *car37* and *the entity observed by the camera*. Some examples of features are the *velocity* of an object, the *road segment* of a vehicle, and the *distance between* two car objects.

A *fluent stream* is a partial representation of a fluent, where a stream of *samples* of the value of the feature at specific time-points is seen as an approximation of the fluent. Due to inherent limitations in sensing and processing, an agent cannot always expect access to the actual value of a feature over time. A sample can either come from an observation of the feature or a computation which results in an estimation of the value at the particular time-point, called the *valid time*. The time-point when a sample is made available or added to a fluent stream is called the *available time*. A fluent stream has certain properties such as start and end time, sample period and maximum delay. These properties are specified by a declarative policy which describes constraints on the fluent stream.

For example, the position of a car can be modeled as a feature. The true position of the car at each time-point during its existence would be its fluent and a particular sequence of observations or estimations of its position would be a fluent stream. There can be many fluent streams all approximating the same fluent.

8.4 State Generation

One important functionality in DyKnow is state generation. Many functionalities require access to a consistent "state of the world", but sensor readings take time to propagate through a distributed system which may consist of multiple AVs together with ground stations and associated hardware and software. DyKnow therefore 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.

For example, if a car has both a speed and a position, then there are two features: "speed of car" and "position of car". But this could also be represented by a single fluent stream containing *tuples* (in this case pairs) of values, called *states*, containing both the speed and the position.

8.5 Semantic Integration of Symbolic Reasoning

One important use of DyKnow is to support the integration of symbolic reasoning in a UAS. To do symbolic reasoning it is necessary to map symbols to streams available in a UAS, which provides them with the intended meaning for the particular UAS.

For example, a temporal logic formula consists of symbols representing variables, sorts, objects, features, and predicates besides the symbols which are part of the logic. Consider $\forall x \in \mathsf{AV} : x \neq \mathsf{av1} \rightarrow \Box\,\mathsf{XYDist}[x, \mathsf{av1}] > 10$, which has the intended meaning that all AVs, except av1, should always be more than 10 meters away from av1. This formula contains the variable x, the sort AV, the object av1, the feature XYDist, the predicates $\neq$ and $>$, and the constant value 10, besides the logical symbols. To evaluate such a formula, see Section 9.3 for details, an interpretation of its symbols must be given. Normally, their meanings are predefined. However, in the case of reasoning over streams the meaning of features cannot be predefined since information about them becomes incrementally available. Instead their meaning has to be determined at run-time. To evaluate the truth value of a formula it is therefore necessary to map feature

symbols to streams, synchronize these streams and extract a state sequence where each state assigns a value to each feature.

In a system consisting of streams, a natural approach is to syntactically map each feature to a single stream. This is called *syntactic integration*. This works well when there is a stream for each feature and the person writing the formula is aware of the meaning of each stream in the system. However, when systems become more complex and when the set of streams or their meaning changes over time it is much harder for a designer to explicitly state and maintain this mapping. Therefore automatic support for mapping features in a formula to streams in a system based on their semantics is needed. This is called *semantic integration*. The purpose of this matching is for each feature to find one or more streams whose content matches the intended meaning of the feature. This is a form of semantic matching between features and contents of streams. The process of matching features to streams in a system requires that the meaning of the content of the streams is represented and that this representation can be used for matching the intended meaning of features with the actual content of streams.

The same approach can be used for symbols referring to objects and sorts. It is important to note that the semantics of the logic requires the set of objects to be fixed. This means that the meaning of an object or a sort must be determined for a formula before it is evaluated and then may not change. It is still possible to have different instances of the same formula with different interpretations of the sorts and objects.

The goal is to automate the process of matching the intended meaning of features, objects, and sorts to content of streams in a system. Therefore the representation of the semantics of streams needs to be machine readable. This allows the system to reason about which stream content corresponds to which symbol in a logical formula. The knowledge about the meaning of the content of streams needs to be specified by a user, even though it could be possible to automatically determine this in the future. By assigning meaning to stream content the streams do not have to use predetermined names, hard-coded in the system. This also makes the system domain-independent, which implies that it could be used to solve different problems in a variety of domains without reprogramming.

The approach to semantic integration in DyKnow uses semantic web technologies to define and reason about ontologies. Ontologies provide suitable support for creating machine readable domain models (Horrocks, 2008). Ontologies also provide reasoning support and support for semantic mapping which is necessary for the integration of streams from multiple UASs.

The Web Ontology Language (OWL) (Smith et al., 2004) is used to represent ontologies. Features, objects and sorts are represented in an ontology with two different class hierarchies, one for objects and one for features.

To represent the semantic content of streams in terms of features, objects, and sorts, a semantic specification language called SSL_T has been defined (Heintz and Dragisic, 2012). This is used to annotate the semantic content of streams.

Finally, a semantic matching algorithm has been developed which finds all streams which contain information relevant to a concept from the ontology, such as a feature. This makes it possible to automatically find all the streams that are relevant for evaluating a temporal logical formula. These streams can then be collected, fused, and synchronized into a single stream of states over which the truth value of the formula is incrementally evaluated. By introducing semantic mapping between ontologies from different UASs and reasoning over multiple related ontologies it is even possible to find relevant streams distributed among multiple UASs (Heintz and Dragisic, 2012).

8.6 ROS-Based Implementation

The ROS-based implementation of DyKnow consists of three main parts: a stream processing part, a stream reasoning part, and a semantic integration part. Each part consists of a set of components. There are three types of components: engines, managers and coordinators. An *engine* takes a specification and carries out the processing as specified. A *manager* keeps track of related items and provides an interface to these. A

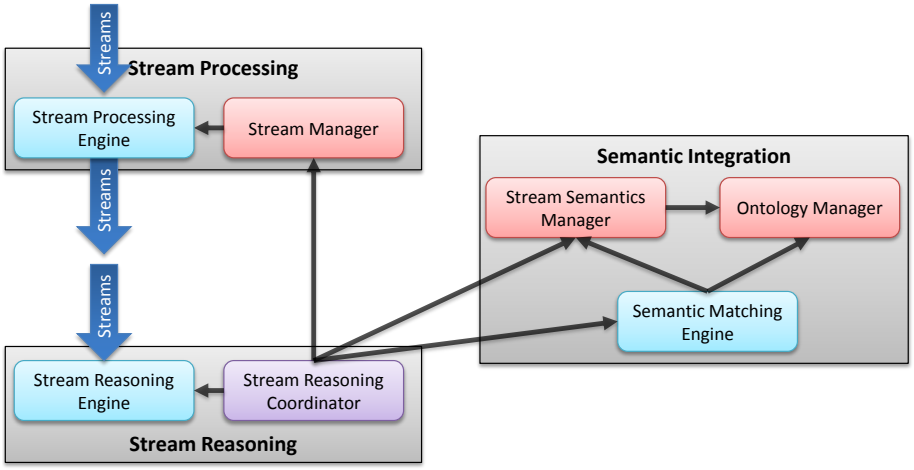


Figure 25: The components of DyKnow.

coordinator provides a high-level functionality by coordinating or orchestrating other functionalities. An overview of the parts and the components is shown in Figure 25. This diagram corresponds to the DyKnow component in Figure 2. The design is very modular as almost every component can be used independently.

The *stream processing* part is responsible for generating streams by for example importing, merging and transforming streams. The *Stream Manager* keeps track of all the streams in the system. Streams can either be generated by a stream processing engine or by some external program. A *Stream Processing Engine* takes a stream specification and generates one or more streams according to the specification.

The *semantic integration* part is responsible for finding streams based on their semantics relative to a common ontology. The *Stream Semantics Manager* keeps track of semantically annotated streams, where an annotation describes the semantic content of a stream. The *Ontology Manager* keeps track of the ontology which provides a common vocabulary. The *Semantic Matching Engine* finds all streams whose semantic annotation matches a particular ontological concept. The semantic integration part is used by the stream reasoning part to find the relevant streams in order to evaluate a logical formula.

The *stream reasoning* part is responsible for evaluating temporal logical formulas over streams as described in Section 9.3. A *Stream Reasoning Engine* takes a logical formula and a stream of states and evaluates the formula over this stream. A *Stream Reasoning Coordinator* takes a logical formula, finds all the relevant streams needed to evaluate the formula, creates a stream specification for generating a single stream of states from all the relevant streams, and instructs the stream reasoning engine to evaluate the formula over the stream as it is generated by a stream processing engine.

8.7 A Traffic Monitoring Example

As a concrete example of the use of DyKnow, Figure 26 provides an overview of how part of the incremental processing required for a traffic surveillance task can be organized as a set of DyKnow knowledge processes.

At the lowest level, a *helicopter state estimation component* uses data from an *inertial measurement unit* (IMU) and a *global positioning system* (GPS) to determine the current position and attitude of the UASTL RMAX. A *camera state estimation component* uses this information, together with the current state of the *pan-tilt unit* on which the cameras are mounted, to generate information about the current camera state. The *image processing component* uses the camera state to determine where the camera is currently pointing. Video streams from the *color* and *thermal cameras* can then be analyzed in order to generate *vision percepts* representing hypotheses about moving and stationary physical entities, including their approximate positions and velocities. The data originating from the PFC and the PPC is provided by the Data Interface through the Platform Server (Section 6.2.1).

Symbolic formalisms such as chronicle recognition (Ghallab, 1996) require a consistent assignment of

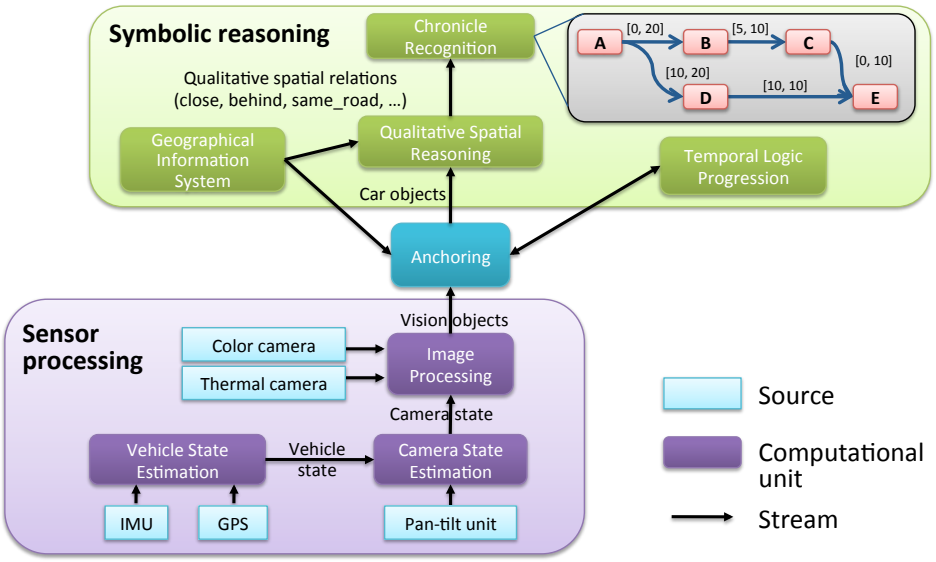


Figure 26: Potential organization of the incremental processing required for a traffic surveillance task.

symbols, or identities, to the physical objects being reasoned about and the sensor data received about those objects. Image analysis may provide a partial solution, with vision percepts having symbolic identities that persist over short intervals of time. However, changing visual conditions or objects temporarily being out of view lead to problems that image analysis cannot (and should not) handle. This is the task of the *anchoring system*, which uses *progression* of formulas in a metric temporal logic to incrementally evaluate potential hypotheses about the observed objects (Heintz et al., 2013). The anchoring system also assists in object classification and in the extraction of higher level attributes of an object. For example, a *geographic information system* can be used to determine whether an object is currently on a road or in a crossing. Such attributes can in turn be used to derive relations *between* objects, including *qualitative spatial relations* such as $\mathsf{beside}(car_1, car_2)$ and $\mathsf{close}(car_1, car_2)$. Concrete events corresponding to changes in such attributes and predicates finally provide sufficient information for the *chronicle recognition system* to determine when higher-level events such as reckless overtakes occur.

9 The Deliberative Layer

Conceptually, the deliberative layer includes all high autonomy functionalities normally associated with rich world models and deep reasoning capabilities. Functionalities commonly associated with the deliberative layer are automated task planners, motion planners, execution monitoring systems, diagnosis systems, and knowledge bases with inference engines, among others. The temporal latencies associated with the decision cycles for these functionalities are often far slower than for functionalities which exist in the reactive and control layers. The decision cycles for the different layers are necessarily asynchronous where each of the sets of functionalities run concurrently. The interfaces between the deliberative layer and the reactive and control layers are essential for transitioning output from deliberative functionality into useful processes at the lower layers of the architecture. In some sense, much of what happens is a form of dynamic compilation which results in transforming qualitative goal-directed assertions into actual actuation commands associated with the control layer of the architecture which contribute to the achievement of these goals.

A deliberative functionality, motion planning, and the dynamic compilation of its output has already been described in Section 7. This section will describe two additional deliberative functionalities central to high autonomy in the HDRC3 architecture. Section 9.2 describes an automated task-based planner, TALplanner, and its integration in the HDRC3 architecture. Section 9.3 describes a novel execution monitoring system based on specifying monitoring queries in terms of temporal logical formulas and evaluating

these formulas on-line in real-time. Due to the complex nature of these functionalities, an open research issue is how one might provide verification and validation techniques for the use and integration of these functionalities, in addition to the formal specification of high-level missions. The section therefore begins with a brief presentation of a temporal action logic (TAL) which provides a formal basis for specifying the semantics of TSTs, TALplanner, and the execution monitor (Section 9.1). Additionally, in Section 9.4, it is shown that high-level missions can in fact be specified in terms of TSTs. Consequently, high-level missions are not only provided with a formal semantics, but due to the relation between these specifications and TSTs, there is a natural means of coupling declarative specification of missions with their procedural execution.

9.1 Temporal Action Logic

Temporal Action Logic, TAL, is a well-established non-monotonic logic for representing and reasoning about actions (Doherty et al., 1998; Doherty and Kvarnström, 2008). This logic provides both clear intuitions and a formal semantics for a highly expressive class of action specifications which supports temporal aspects, concurrent execution, and incomplete knowledge about the environment and the effects of an action. Therefore it is a highly suitable basis for describing and reasonable the elementary actions used in realistic mission specifications. Here a limited subset of TAL is described and the reader is referred to Doherty and Kvarnström (2008) for further details.

TAL provides an extensible macro language, $\mathscr{L}$(ND), that supports the knowledge engineer and allows reasoning problems to be specified at a higher abstraction level than plain logical formulas. The basic ontology includes parameterized features $f(\bar{x})$ that have values v at specific timepoints t, denoted by $[t]f(\bar{x}) \hat{=} v$, or over intervals, $[t,t']f(\bar{x}) \hat{=} v$. Incomplete information can be specified using disjunctions of such facts. Parameterized actions can occur at specific intervals of time, denoted by $[t_1,t_2]A(\bar{x})$. To reassign a feature to a new value, an action uses the expression $R([t]f(\bar{x}) \hat{=} v)$. Again, disjunction can be used inside $R()$ to specify incomplete knowledge about the resulting value of a feature. The value of a feature at a timepoint is denoted by $value(t,f)$.

The logic is based on scenario specifications represented as *narratives* in $\mathscr{L}$(ND). Each narrative consists of a set of statements of specific types, including *action type specifications* defining named actions with preconditions and effects. The basic structure, which can be elaborated considerably (Doherty and Kvarnström, 2008), is as follows:

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

stating that if the action $A(\bar{v})$ is executed during the interval $[t_1,t_2]$, then given that its preconditions $\Gamma_{pre}(t_1,\bar{v})$ are satisfied, its effects, $\Gamma_{post}(t_1,t_2,\bar{v})$, will take place. Additionally, $\Gamma_{cons}(t_1,t_2,\bar{v})$ can be used to specify logical constraints associated with the action. For example, the following defines the elementary action **fly-to**: If a AV should fly to a new position (x',y') within the temporal interval $[t,t']$, it must initially have sufficient fuel. At the next timepoint $t+1$ the AV will not be hovering, and in the interval between the start and the end of the action, the AV will arrive and its fuel level will decrease. Finally, there are two logical constraints bounding the possible duration of the flight action.

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

The translation function $Trans()$ translates $\mathscr{L}$(ND) expressions into $\mathscr{L}$(FL), a first-order logical language

(Doherty and Kvarnström, 2008). This provides a well-defined formal semantics for narratives in $\mathscr{L}$(ND). This separation between the macro language and the base logic makes TAL highly extensible. When adding new constructs to the formalism, new expression types are defined in $\mathscr{L}$(ND) and $Trans()$ is extended accordingly, generally with no extension required in the base language $\mathscr{L}$(FL).

The $\mathscr{L}$(FL) language is order-sorted, supporting both types and subtypes for features and values. This is also reflected in $\mathscr{L}$(ND), where one often assumes variable types are correlated to variable names – for example, av_3 implicitly ranges over AVs. There are a number of sorts for values $\mathscr{V}_i$, including the Boolean sort $\mathscr{B}$ with the constants $\{true, false\}$. $\mathscr{V}$ is a supersort of all value sorts. There are a number of sorts for features $\mathscr{F}_i$, each one associated with a value sort $dom(\mathscr{F}_i) = \mathscr{V}_j$ for some j. The sort $\mathscr{F}$ is a supersort of all fluent sorts. There is also an action sort $\mathscr{A}$ and a temporal sort $\mathscr{T}$. Generally, t, t' will denote temporal variables, while $\tau, \tau', \tau_1, \ldots$ are temporal terms. $\mathscr{L}$(FL) currently uses the following predicates, from which formulas can be defined inductively using standard rules, connectives and quantifiers of first-order logic.

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

When a narrative is translated, $Trans()$ first generates the appropriate $\mathscr{L}$(FL) formulas corresponding to each $\mathscr{L}$(ND) statement. Foundational axioms such as unique names and domain closure axioms are appended when required. Logical entailment then allows the reasoner to determine when actions must occur, but the fact that they *cannot* occur at other times than explicitly stated is not logically entailed by the translation. This problem is handled in a general manner through filtered circumscription, which also ensures that fluents can change values only when explicitly affected by an action or dependency constraint (Doherty and Kvarnström, 2008).

The structure of $\mathscr{L}$(ND) statements ensures that the second-order circumscription axioms are reducible to equivalent first-order formulas, a reduction that can often be performed through predicate completion. Therefore, classical first-order theorem proving techniques can be used for reasoning about TAL narratives (Doherty and Kvarnström, 2008). For unmanned systems, however, the logic will primarily be used to ensure a correct semantics for planners, execution monitors and mission specification languages and correlating this semantics closely to the implementation. Using TAL does not require theorem proving on board.

9.2 Task Planning using TALplanner

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 in the HDRC3 architecture: Elementary action nodes in TSTs provide a set of high-level actions such as "take off" and "fly to point A", but one also makes use of automated planning techniques to *compose* such actions into plans satisfying a set of declaratively specified goals. The goals themselves will vary depending on the nature of a mission. For example, they could involve having acquired images of certain buildings or having delivered crates of

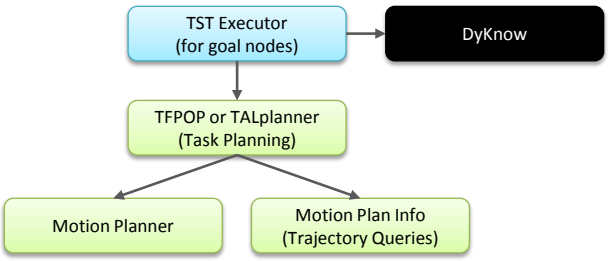


Figure 27: Task planning called by TST executors for goal nodes.

emergency supplies to certain locations after a natural disaster.

Support for planning is closely integrated in the HDRC3 architecture through the use of *goal nodes* in Task Specification Trees (Section 6.3). Any given TST can contain multiple goal nodes, each of which describes a simple or complex goal to be achieved at that particular point in the corresponding mission. When a goal node is assigned to a specific agent, the executor for that goal node can call DyKnow (Section 8) or use any other means at its disposal to acquire essential information about the current state of the world and the AV's own internal state (Figure 27). The node is then *expanded* by that agent through a call to an onboard task planner, which in turn can call the motion planner and Motion Plan Info functionalities discussed earlier (Section 7.1). The resulting concurrent plan is converted to a new "plan sub-tree" attached under the goal node. Since TST structures are inherently distributable, the entire plan can then be distributed to the participating agents. This is the principal means of invoking a task planning functionality.

Note that the resulting TST directly represents a plan using sequence and concurrency nodes together with elementary action nodes corresponding to plan actions. Therefore a plan can be executed at the appropriate time in the standard manner using TST executors. This obviates the need for a separate plan execution system and results in an integrated means of executing a mission, regardless of the techniques or combination of techniques that was used to generate it.

The planner that is used to expand a goal node can be changed freely, and in a heterogeneous system different platforms may use different planners. Currently the planners most commonly used in the HDRC3 architecture are TALplanner (Doherty and Kvarnström, 2001; Kvarnström, 2005) and TFPOP (Kvarnström, 2011), two domain-independent concurrent temporal planners that were developed explicitly for use in this architecture. These planners are similar in certain respects: Both have a semantics based on Temporal Action Logic (Section 9.1), and both use *domain-specific control formulas* in this logic to guide the search for a solution, leading to very rapid plan generation. However, they use different search spaces and plan structures, and they differ in the expressivity of the temporal constraints they permit. TALplanner will be described here, while the reader is referred to Kvarnström (2011) for further information about TFPOP.

TALplanner. TALplanner (Doherty and Kvarnström, 1999; Kvarnström, 2005; Kvarnström and Doherty, 2000) is a forward-chaining temporal concurrent planner where planning domains and problem instances are specified as *goal narratives* in a version of TAL extended with new macros and statement types for plan operators, resource constraints, goal specifications, and other types of information specific to the task of plan generation. These macros serve to simplify planning domain specifications, but retain a basis in standard TAL through an extended translation function. For example, plan operators are converted into standard TAL action type specifications, and actions in a plan are represented as standard timed action occurrences of the form $[t_1, t_2]\, A(\bar{v})$.

Control Formulas. In addition to providing a declarative first-order semantics for planning domains, TAL is also used to specify a set of domain-specific temporal *control formulas* acting as constraints on the set of valid solutions: A plan is a *solution* only if its final state satisfies the goal *and* all control formulas are satisfied in the complete state sequence that would result from executing the plan, which can be viewed as

a logical model.

The use of control formulas serves two separate purposes. First, it allows the specification of complex temporally extended goals such as safety conditions that must be upheld throughout the (predicted) execution of a plan. Second, the additional constraints on the final solution often allow the planner to prune entire branches of the search tree – whenever it can be proven that every search node on the branch corresponds to a state sequence that violates at least one control rule. Formulas can then be written to guide the planner towards those parts of the search space that are more likely to contain plans of high quality, in terms of time usage or other quality measures.

As an example, consider three simple control rules that could be used in an airplane-based logistics domain. First, a package should only be loaded onto a plane if a plane is required to move it: If the goal requires it to be at a location in another city. Regardless of which operator is used to load a package, one can detect this through the fact that it is in a plane at time $t+1$, but was *not* in the same plane at time t.

$\forall t, obj, plane, loc.$
[t] $\neg$in(obj, $plane$) $\wedge$ at(obj, loc) $\wedge$ [t+1] in(obj, $plane$) $\rightarrow$
$\exists loc'$ [goal (at(obj, loc')) $\wedge$ [t] city_of(loc) $\not\hat{=}$ city_of(loc')]

Second, if a package has been unloaded from a plane, there must be a valid reason for this: It must be the case that the package should be in the city where the plane has landed.

$\forall t, obj, plane, loc.$
[t] in(obj, $plane$) $\wedge$ at($plane$, loc) $\wedge$ [t+1] $\neg$in(obj, $plane$) $\rightarrow$
$\exists loc'$ [goal (at(obj, loc')) $\wedge$ [t] city_of(loc) $\hat{=}$ city_of(loc')]

Third, if a package is at its destination, it should not be moved.

$\forall t, obj, loc.$
[t] at(obj, loc) $\wedge$ goal (at(obj, loc)) $\rightarrow$ [t+1] at(obj, loc)

Surprisingly, such simple hints to an automated planner can often improve planning performance by orders of magnitude given that the planner has the capability to make use of the hints.

Concurrent Plan Structure. Forward-chaining planning is most often used for sequential plans, where each new action is added immediately after the previous one in the plan. TALplanner uses a similar technique for concurrent plan generation, with a relaxed constraint on where a new action occurrence is placed.

Definition 9.1 (Concurrent Plan) *A* concurrent plan *for a goal narrative $\mathscr{G}$ is a tuple of ground fluent-free action occurrences with the following constraints. First, the empty tuple is a concurrent plan for $\mathscr{G}$. Second, given a concurrent plan $p = \langle [\tau_1, \tau_1'] \; o_1(\bar{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n(\bar{c}_n) \rangle$ for $\mathscr{G}$, its successors are the sequences that add one new action occurrence $[\tau_{n+1}, \tau_{n+1}'] \; o_{n+1}(\bar{c}_{n+1})$ and satisfy the following constraints:*

1. *Let $\mathscr{G}' = \mathscr{G} \cup \{ [\tau_1, \tau_1'] \; o_1(\bar{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n(\bar{c}_n) \}$ be the original goal narrative $\mathscr{G}$ combined with the existing plan. Then, the new action $o_{n+1}(\bar{c}_{n+1})$ must be applicable over the interval $[\tau_{n+1}, \tau_{n+1}']$ in $\mathscr{G}'$. This implies that its preconditions are satisfied, that its effects are not internally inconsistent and do not contradict the effects of the operator instances already present in the sequence, and that the duration $\tau_{n+1}' - \tau_{n+1}$ is consistent with the duration given in the operator specification.*
2. *$\tau_1 = 0$: The first action starts at time 0.*
3. *$\tau_{n+1} \geq \tau_n$: The new action cannot be invoked before any of the actions already added to the plan.*
4. *$\tau_{n+1} \leq \max(\tau_1', \ldots, \tau_n')$: There can be no gap between the time interval covered by the current plan and the start of the newly added action.*

Algorithm 3 Concurrent TALplanner.

Input: A goal narrative $\mathscr{G}$.

Output: A plan narrative entailing the goal $\mathscr{N}_{\text{goal}}$ and the control formulas $\mathscr{N}_{\text{control}}$.

```
1  procedure TALplanner-concurrent(𝒢)
2  γ ← ⋀𝒢_goal                                          Conjunction of all goal statements
3  ⟨init, incr, final⟩ ← generate-pruning-constraints(𝒢_control)
4  node ← ⟨init, ∅, 0, 0, ⟨⟩⟩                          ⟨cond. queue, visited states, latest invocation time, t_max, plan⟩
5  Open ← ⟨node⟩                                        Stack (depth first search)
6  while Open ≠ ⟨⟩ do
7    ⟨C, S, τ_0, τ_max, p⟩ ← pop(Open)                  Current plan candidate
8    𝒢' ← 𝒢 ∪ p ∪ occlude-all-after(𝒢, τ_0)             No knowledge about future
9    for all constraints α in C do                      Check queued constraints
10     if Trans⁺(𝒢') ⊨ Trans(α) then C ← C \ {α}         Remove satisfied constraint
11     elsif Trans⁺(𝒢') ⊨ Trans(¬α) then backtrack      Constraint violated
12   𝒢'' ← 𝒢 ∪ p ∪ {t_max = τ_max}                      Narrative with complete knowledge
13   if Trans⁺(𝒢'') ⊨ false then backtrack              Consistency check
14   if ∃s ∈ S.better-or-equal(s, final state of current plan) then backtrack
15   if Trans⁺(𝒢'') ⊨ Trans(γ ∧ C ∧ final) then         Goal + queued + final ctrl satisfied
16     return 𝒢''
17   else                                               Not a solution, but check children
18     S' ← S ∪ {final state of current plan}
19     for all successor actions A = [ρ, ρ'] o^i(c̄) for p according to Def 9.1 do
20       C' ← C ∪ incr_i[ρ, c̄]                          Old conditions + incr control
21       C' ← C' ∪ {prevail condition of A}             Add prevail condition
22       push ⟨C', S', ρ, max(τ_max, ρ'), ⟨p; A⟩⟩ onto Open
23 fail
```

Generating Concurrent Plans. The concurrent TALplanner algorithm will now be briefly described (Algorithm 3). The reader is referred to Kvarnström (2005) for further details.

First, the planner conjoins all goal statements (line 2) and uses the control formulas in the goal narrative to generate a set of *pruning constraints*. These constraints allow control formulas to be verified efficiently when actions are incrementally added to a plan. Specifically, each control formula may result in an *initial constraint* to be tested initially, a set of operator-specific *incremental constraints* to be tested whenever an instance of a particular operator is added to a plan, and a *final* constraint to be tested in the final solution.

The initial search node is created (line 4), as is a stack of open search nodes (for depth first search).

As long as the search space has not been exhausted (line 6), the planner retrieves the topmost node in the stack of open nodes. This node consists of a queue of constraints that remain to be evaluated (C), a set of visited states to be used in cycle checking (S), the latest invocation time of any action in the current plan (τ_0), the latest end time of any action in the current plan (t_{max}), and the current plan candidate (p).

Assuming a completely specified initial state and deterministic actions, the narrative $\mathscr{G} \cup p$ would now contain complete information about the development of the world that would result from executing exactly the plan p and nothing else. This then corresponds to a unique infinite state sequence specifying the values of all TAL fluents at all points in time. However, the current search node must only be pruned if *all possible extensions* of p violate a control formula. Given the constraints placed on successors in Definition 9.1, no action added to p can be invoked earlier than τ_0 and therefore all effects of actions added to p must take place at $\tau_0 + 1$ or later. Line 8 therefore generates the narrative $\mathscr{G} \cup p \cup \text{occlude-all-after}(\mathscr{G}, \tau_0)$ which has additional formulas disclaiming knowledge about fluents after time t_0.

In lines 9–11, the planner determines whether a queued constraint is now definitely true (in which case it can be removed from the queue) or definitely false (in which case the planner has to backtrack).

Then, the planner must verify certain conditions under the assumption that no additional actions are

added. A narrative $\mathscr{G}''$ assuming complete information is therefore generated in line 12. If this is inconsistent (which in practice can be tested efficiently given the expressivity of TALplanner operators), the planner must backtrack (line 13). If a state that is better or equal has already been visited, the planner should also backtrack (line 14). The better-or-equal relation can for example take into account resource availability, where a state that is equal in all respects except that it provides more of a particular resource can be considered strictly better.

In lines 15–16, TALplanner determines whether the current plan candidate satisfies the goal and all remaining pruning constraints. If this is the case, a solution is found and can be returned. If not, the node is expanded through the generation of a set of successors. Each successor may be given additional incremental pruning constraints, instantiated with the actual invocation timepoints and actual arguments of the corresponding action (line 20). Similarly, any *prevail conditions* (similar to preconditions but relating to the entire execution interval of an action) are added to the condition queue, to be tested when more information about future states is available.

9.3 Execution Monitoring

Regardless of the effort spent modeling all possible contingencies, actions and plans may fail. Robust performance in a noisy environment therefore requires some form of supervision, where the execution of a plan is constantly monitored in order to detect and recover from potential or actual failures. For example, since an AV might accidentally drop its cargo, it should monitor the condition that whenever it carries a crate, the crate remains until the AV 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 AV 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. While some of these constraints would best be monitored at the control level, there are also many cases where monitoring and recovery should be lifted into a higher level *execution monitor* (Ben Lamine and Kabanza, 2002; De Giacomo et al., 1998; Fernández and Simmons, 1998; Fichtner et al., 2003; Gat et al., 1990).

The execution monitoring system described here is based on an intuition similar to the one 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 by DyKnow (Section 8). Each new sampling point generates a new state which provides information about all state variables used by the current monitor formulas, thereby providing information about the *actual* state of the world as opposed to what could be *predicted* from the domain specification. The resulting sequence of states corresponds to a partial logical interpretation, where "past" and "present" states are completely specified whereas "future" states are completely undefined.

Note that simply comparing the actual and predicted states and signaling a violation as soon as a discrepancy is found is not sufficient, because not all discrepancies are fatal – for example, if the altitude was predicted to be 5.0 meters and the current measurement turns out to be 4.984 meters. Also, some information might be expensive or difficult to sense, in which case the ground operator should be given more control over which information is actually gathered and used for monitoring. Sensing may even require special actions that interfere with normal mission operations. Finally, the richer the planner's domain model is, the more it 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. Therefore, most conditions to be monitored are *explicitly* specified, though many conditions *can* be automatically generated within the same framework if so desired.

Execution Monitor Formulas in TAL. Execution monitor formulas are expressed in a variation of TAL where the high-level language $\mathscr{L}$(ND) is augmented with a set of *tense operators* similar to those used in modal tense logics such as LTL (Emerson, 1990) and MTL (Koymans, 1990). Tense operators allow

the expression of complex metric temporal conditions and are amenable to incremental evaluation as each new state is generated. This allows violations to be detected as early and as efficiently as possible using a *formula progression* algorithm, while the basis in TAL provides a common formal semantic ground for planning and monitoring.

Three tense operators have been introduced into $\mathscr{L}$(ND): U (until), $\Diamond$ (eventually), and $\Box$ (always). Like all expressions in $\mathscr{L}$(ND), these operators are macros on top of the first order base language $\mathscr{L}$(FL).

Definition 9.2 (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,*
- $\omega \leq \omega'$, $\omega < \omega'$, *or* $\omega = \omega'$, *where* ω *and* ω' *are value terms,*
- *f, where f is a boolean fluent term (state variable term),*
- $f \hat{=} \omega$, *where f is a fluent term and* ω *is a value term of the corresponding sort,*
- $\phi \, \mathsf{U}_{[\tau,\tau']} \, \psi$, *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, or*
- *a combination of monitor formulas using 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.*

Tense operators use relative time, where each formula is evaluated relative to a "current" timepoint. The semantics of these formulas satisfies the following conditions (see Doherty et al. (2009) for details):

- 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 $\mathtt{true} \, \mathsf{U}_{[\tau,\tau']} \, \phi$ 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 $t' \in [t+\tau, t+\tau']$.

Example 9.1 *Suppose that an AV 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:*

$\Box \forall av.(\mathit{power}(av) > M \rightarrow \mathit{power}(av) < f \cdot M \, \mathsf{U}_{[0,\tau]} \, \Box_{[0,\tau']} \, \mathit{power}(av) \leq M)$ □

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

Monitors and Actions. In many cases, conditions that should be monitored are closely related to the actions currently being executed. Two extensions are made to facilitate the specification of such formulas.

First, monitor formulas can 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.

Example 9.2 *Execution should be monitored when an AV attempts to pick up a box. Since the arguments of pickup-box include the av and the box, the following operator-specific monitor formula can be used:*

$\Diamond_{[0,5000]} \Box_{[0,1000]}$ *carrying*(av, box)

Within 5000 ms, the AV should detect that it is carrying the box, and it should detect this for at least 1000 ms. The latter condition protects against problems during the pickup phase, where the box may be detected during a very short period of time even though the ultimate result is failure. □

Second, a set of *execution flags* allow monitor formulas to query the internal execution state of an agent. This is useful when one wants 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 or *before* the execution of another action. For example, if an AV *picks up a crate*, it should sense the weight of the crate until it deliberately *puts down the crate*.

An execution flag is an ordinary boolean state variable which holds exactly when the corresponding action is being executed. By convention, this state variable will generally be named by prepending "executing-" to the name of the corresponding operator: The pickup-box operator is associated with the executing-pickup-box execution flag, which takes a subset of the operator's parameters. TST executors for elementary action nodes can set this flag when execution starts and clear it when execution ends.

Example 9.3 *Consider the climb(av) operator, which should cause the AV to ascend to its designated flight altitude. Here, one may wish to monitor the fact that the AV truly ends up at its flight altitude. This can be achieved using the formula* executing-climb(av) U altitude$(av) \geq 7.0$. □

When it is clear from context which operator is intended, the shorthand notation EXEC can be used to refer to its associated execution flag with default parameters: EXEC U altitude$(av) \geq 7.0$.

Example 9.4 *Whenever an AV picks up a box, it should detect the box within 5000 ms and keep detecting it until it is explicitly put down. Using an operator-specific monitor formula for pickup-box:*

EXEC $U_{[0,5000]}$(carrying(av, box) U executing-putdown(av, box))

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. For example, one can automatically generate formulas verifying that preconditions and prevail conditions (which must hold throughout the execution of an action) are satisfied, that expected effects take place, and that bounds on action durations are not violated. Similarly one can automatically extract *causal links*, where one action achieves one condition that is later needed by another, and generate a formula verifying that this condition is never violated in the interval between these actions. Since there are pragmatic reasons for *not* generating all possible monitor formulas, automatic generation only takes place for those conditions that are flagged for monitoring. This provides the benefits of automatic formula generation while keeping the control in the hands of the domain designer. See Doherty et al. (2009) for details.

9.3.1 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 autonomous flight. Goals and constraints may be time-dependent, making local repairs difficult: An AV might only be allowed to fly in certain areas at certain times. Also, recovering from a failure for one AV may require changing the plans of another AV: If **heli1** fails to deliver a box of medicine on time, **heli2** might have to be rerouted. Therefore the main focus has been on recovery through replanning. Given that the planner is sufficiently fast when generating new plans,

this does not adversely affect the execution of a fully autonomous mission.

Having detected a failure, the first action of an AV is to suspend the execution of the current plan, execute an emergency break if required, and then go into autonomous hover mode. Currently, one takes advantage of the fact that the UAS Tech Lab RMAX and LinkQuads are 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 an AV fails to take off with a certain carrier, the associated recovery operator can adjust the AV's assumptions about how many boxes it is able to lift. This feeds 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 AV 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.

9.3.2 Execution Monitoring with Inaccurate Sensors

Monitoring should not only maximize the probability that a failure is detected but also minimize the probability of false positives, where a failure is signaled but none has occurred. Some problems, such as those caused by dropouts and communication delays, can be ameliorated by extrapolating historical values and by delaying state generation slightly to allow values to propagate through the distributed system. Noise could be minimized through sensor value smoothing techniques and sensor fusion techniques. However, inaccuracies in the detected state sequence can never be completely eliminated. Therefore, the fact that state values may be inaccurate should be taken into consideration when writing monitor formulas.

For example, the meaning of the condition $\Box \forall av.\mathsf{speed}(av) \leq T$ is that the *sensed and approximated* speed of an AV must never exceed the threshold T. Since a single observation of $\mathsf{speed}(av)$ 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 $\Box \Diamond_{[0,\tau]} \mathsf{speed}(av) \leq T$: It should always be the case that within the interval $[0,\tau]$ from now, the sensed speed returns to being below the threshold.

This formula is somewhat weak: It only requires that a single measurement in every interval of length τ is below the threshold. An alternative would be to require that within τ time units, there will be an *interval* of length τ' during which the AV stays within the limits: $\Box(\mathsf{speed}(av) > T \rightarrow \Diamond_{[0,\tau]} \Box_{[0,\tau']} \mathsf{speed}(av) \leq T)$.

9.3.3 Formula Progression

To promptly detect violations of monitor conditions during execution, a *formula progression* algorithm is used (Bacchus and Kabanza, 1998). 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 $[s_1, \ldots, s_n]$. In essence, this evaluates those parts of the monitor formula that refer to s_0, returning a new formula to be progressed in the same manner once s_1 arrives.

If the formula $\perp$ (false) is returned, then sufficient information has been received to determine that the monitor formula must be violated regardless of the future development of the world. For example, this will happen as soon as the formula $\Box\,\mathsf{speed} < 50$ is progressed through a state where $\mathsf{speed} \geq 50$. Similarly, if $\top$ (true) is returned, 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 complies 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.

Definition 9.3 (Progression of Monitor Formulas) *The following algorithm is used for progression of*

monitor formulas. Note that states are not first-class objects in TAL *and are therefore identified by a timepoint τ and an interpretation $\mathscr{I}$. Special cases for $\Box$ and $\Diamond$ can be introduced for performance.*

```
1  procedure Progress(φ, τ, I)
2  if φ = f(x̄) ≙ v
3    if I ⊨ Trans([τ] φ) return ⊤ else return ⊥
4  if φ = ¬φ1 return ¬Progress(φ1, τ, I)
5  if φ = φ1 ⊗ φ2 return Progress(φ1, τ, I) ⊗ Progress(φ2, τ, I)
6  if φ = ∀x.φ // where x belongs to the finite domain X
7    return ⋀_{c∈X} Progress(φ[x ↦ c], τ, I)
8  if φ = ∃x.φ // where x belongs to the finite domain X
9    return ⋁_{c∈X} Progress(φ[x ↦ c], τ, I)
10 if φ contains no tense operator
11   if I ⊨ Trans(φ) return ⊤ else return ⊥
12 if φ = φ1 U_{[τ1,τ2]} φ2
13   if τ2 < 0 return ⊥
14   elsif 0 ∈ [τ1, τ2] return Progress(φ2, τ, I) ∨ (Progress(φ1, τ, I) ∧ (φ1 U_{[τ1−1,τ2−1]} φ2))
15   else return Progress(φ1, τ, I) ∧ (φ1 U_{[τ1−1,τ2−1]} φ2)
```

The result of Progress is 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 $\Diamond_{[0,\tau]}\,\phi \wedge \Diamond_{[0,\tau']}\,\phi \equiv \Diamond_{[0,\min(\tau,\tau'')]}\,\phi$.

For this 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. The following evaluation uses one synthetic test where one can study complex combinations of time and modality. An earlier version of the actual DRC computer on board a UASTL RMAX was used to run progression tests for formulas having a form that is typical for monitor formulas in many applications. State sequences are constructed to exercise both the best and the worst cases for these formulas.

The evaluation shows that even with the worst possible inputs, complex formulas can be evaluated in less than 1 millisecond per state and formula on this CPU (1.4 GHz Pentium M). One example formula is $\Box\neg p \rightarrow \Diamond_{[0,1000]}\Box_{[0,999]}\,p$, corresponding to the fact that if the property p is false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true. For example, the previously discussed formula $\Box(\mathsf{speed}(av) > T \rightarrow \Diamond_{[0,\tau]}\Box_{[0,\tau']}\,\mathsf{speed}(av) \leq T)$ has this general form. To estimate the cost of evaluating this formula, it was progressed through several different state streams corresponding to the best case, the worst case, and two intermediate cases. A new state in the stream was generated every 100 ms, which means that all formulas must be progressed within this time limit or monitoring will fall behind. Figure 28 shows the average time required to progress a certain number of formulas through each state in a sequence. This indicates that 100 ms is sufficient for progressing between 1500 and 3000 formulas of this form on the computer on-board the UASTL RMAX, depending on the state stream. Similar results have been shown for formulas of different forms, such as $\Box\neg p \rightarrow \Diamond_{[0,1000]}\Box_{[0,999]}\,p$ and $\Box\neg p \rightarrow \Diamond_{[0,1000]}\Box_{[0,999]}\,p$ (Doherty et al., 2009).

9.4 High-Level Mission Specification

The ability to clearly and concisely specify missions is fundamentally important for an unmanned system to be practically useful, and requires a suitable *mission specification language* that should satisfy a variety of requirements and desires. The language should be comprehensible to humans and not only useful as an intermediate representation both generated and received by software. At the same time intuitions are not sufficient: A strict formal semantics must be available. There should also be a close connection to mission

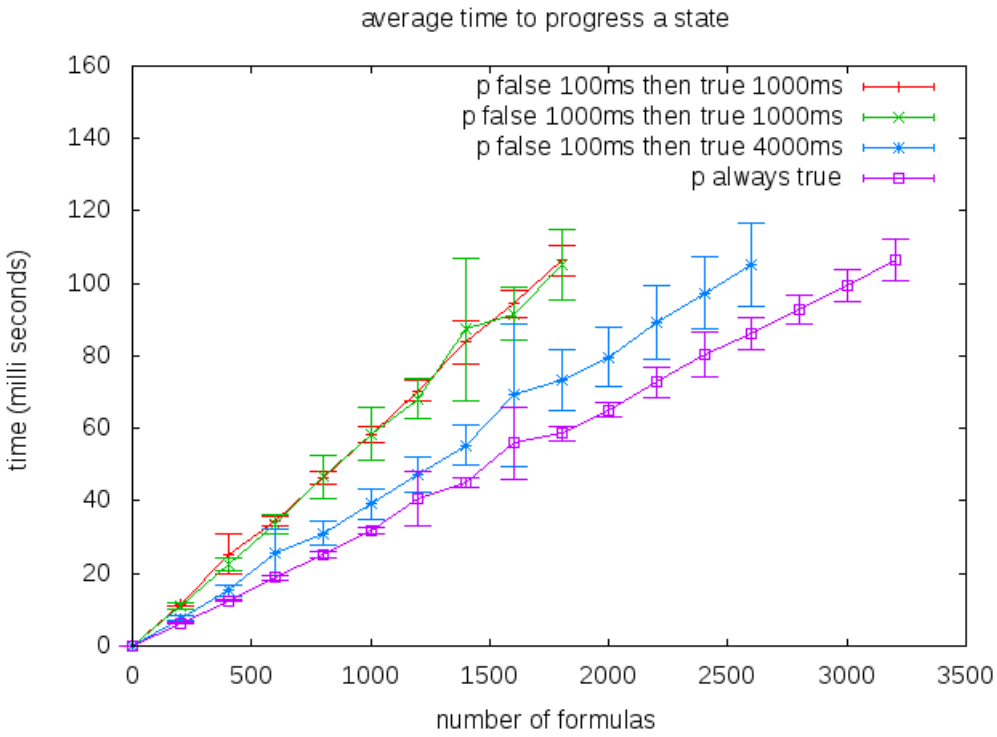


Figure 28: Average progression time: Always $\neg p \rightarrow$ Eventually Always p.

execution, allowing the actual semantics of the language to be used to specify the correct operation of the system and thereby facilitating the validation of the system as a whole. A principled foundation where these issues are considered, for single platforms (vehicles) as well as for fleets of homogeneous or heterogeneous platforms, is essential for these types of systems to be accepted by aviation authorities and in society.

The mission specification language used in the HDRC3 architecture is designed to allow partial mission specifications with constraints, including resource requirements and temporal deadlines. It extends the support for highly expressive *elementary actions* in Temporal Action Logic (TAL, Section 9.1) with *temporal composite actions* that also provide a formal semantics for the high-level structure of a mission (Doherty et al., 2012). The composite action constructs in the extended logic TALF correspond closely to the control structures supported by Task Specification Trees (Section 6.3). As shown in Figure 29, missions can therefore be defined either in TALF or as TSTs and can be translated in either direction. The *delegation* functionality discussed in Section 10 can then be used to delegate a mission in the shape of a TST to one or more platforms for execution, resulting in a distributed task structure.

Composite actions in TALF are characterized recursively through the general construct "with VARS do TASK where CONS": Any composite action consists of a task TASK that should be executed in a context characterized by a set of variables VARS constrained by a set of constraints CONS. The TASK, in turn, can be an elementary TAL action or consist of a combination of composite actions using constructs such as sequential composition, parallel (concurrent) composition, conditionals, (unbounded) loops, while-do, and a concurrent for-each operator allowing a variable number of actions to be executed concurrently. At the mission specification level considered here, each constraint definition can be as general as a logical formula in TAL, giving it a formal semantics. For pragmatic use in a robotic architecture, a wide class of formulas can be automatically transformed into constraints processed by a constraint satisfaction solver, allowing a robotic system to formally verify the consistency of a (distributed) task through the use of (distributed) constraint satisfaction techniques.

A *composite action type specification* declares a named composite action. This is useful in order to

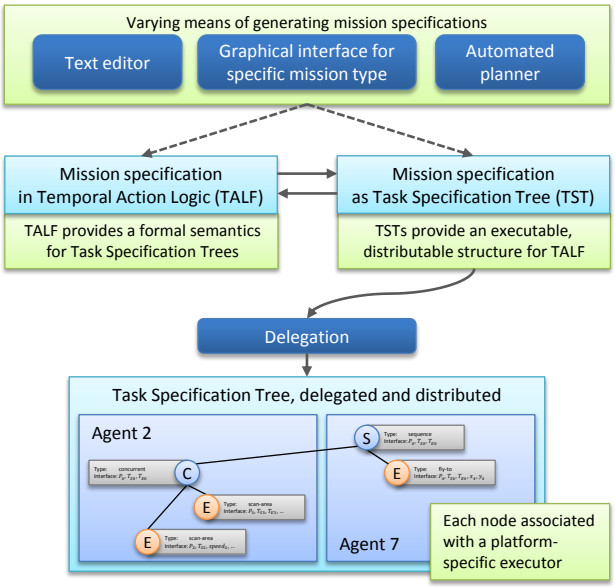


Figure 29: Mission specification, translation and delegation.

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

$$
\begin{array}{lll}
\text{C-ACT} & ::= & [\tau,\tau'] \text{ with } \bar{x} \text{ do TASK where } \phi \\
\text{TASK} & ::= & [\tau,\tau'] \text{ ELEM-ACTION-TERM} \mid \\
 & & [\tau,\tau'] \text{ COMP-ACTION-TERM} \mid \\
 & & (\text{C-ACT; C-ACT}) \mid \\
 & & (\text{C-ACT} \mid\mid \text{C-ACT}) \mid \\
 & & \text{if } [\tau]\ \psi \text{ then C-ACT else C-ACT} \mid \\
 & & \text{while } [\tau]\ \psi \text{ do C-ACT} \mid \\
 & & \text{foreach } \bar{x} \text{ where } [\tau]\ \psi \text{ do conc C-ACT}
\end{array}
$$

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

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

$$
\begin{array}{l}
[t_1,t_2]\text{with } av,t_3,t_4,t_5,t_6 \text{ do} \\
\quad \big([t_3,t_4]\textbf{fly-to}(av,x,y);[t_5,t_6]\textbf{collect-video}(av,x,y)\big) \\
\text{where } [t_1]\text{has-camera}(av)
\end{array}
$$

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

The with-do-where construct provides a flexible means of constraining variables as desired for the task

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

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

Formal Semantics. To define a formal semantics for TALF, the base logic $\mathscr{L}$(FL) is first extended with support for *fixpoint formulas*. This is required for unbounded loops and unbounded recursion, which cannot be characterized in the first-order logic $\mathscr{L}$(FL). A translation is then defined from TALF into the extended base logic $\mathscr{L}(\mathrm{FL}_{\mathrm{FP}})$. As a result of this translation, a composite action is a theory in $\mathscr{L}(\mathrm{FL}_{\mathrm{FP}})$. Questions about missions thereby become queries relative to an inference mechanism, allowing operators to analyze mission properties both during pre- and post-mission phases. This also provides a formal semantics for Task Specification Trees, which can be translated into TALF, and thereby for the execution system used in the HDRC3 architecture. Details regarding translations and the resulting formal semantics are specified in Doherty et al. (2012).

Examples. In November 2011, a powerful earthquake off the coast of Japan triggered a tsunami with devastating effects, including thousands of dead and injured as well as extensive damage to cities and villages. Another effect, which became increasingly apparent over time, was the extensive damage to the Fukushima Daiichi nuclear plant which later resulted in a complete meltdown in three reactors. The exact level of damage was initially difficult to assess due to the danger in sending human personnel into such conditions. Here unmanned aircraft could immediately have assisted in monitoring radiation levels and transmitting video feeds from a closer range. Several composite actions that can be useful for the problem of information gathering in this situation will now be considered. The focus is on demonstrating the $\mathscr{L}$(ND) composite action constructs and some aspects of the actions below are simplified for expository reasons.

Assume the existence of a set of elementary actions whose meaning will be apparent from their names and from the explanations below: **hover-at**, **fly-to**, **monitor-radiation**, **collect-video**, and **scan-cell**. Each elementary action is assumed to be defined in standard TAL and to provide suitable preconditions, effects, resource requirements and (completely or incompletely specified) durations. For example, only an AV with suitable sensors can execute **monitor-radiation**.

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

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

The first part of the constraint specified in the *where* clause ensures that an AV involved in a monitoring action is equipped for surveillance and is radiation-hardened (in addition to the conditions placed on **monitor-radiation**, which include the existence of radiation sensors). The temporal constraints model a

requirement for these particular actions to be synchronized in time and for the AV to hover in a stable location throughout the execution of monitor-single. These constraints could easily be relaxed, for example by stating that hovering occurs throughout the action but monitoring occurs in a sub-interval.

The following action places four AVs in a diamond pattern to monitor a given location such as a nuclear reactor at a given distance, counted in grid cells. The AVs involved are not specified as parameters to the monitoring action but are chosen freely among available AVs, subject to the constraints modeled by sub-actions such as monitor-single.

$$
\begin{array}{l}
[t,t']\underline{\text{monitor-pattern}}(x,y,dist) \rightsquigarrow \\
[t,t']\text{with } s_1,\ldots,w_4,av_1,av_2,av_3,av_4 \text{ do } (\\
\quad ([s_1,s_2]\ \textbf{fly-to}(av_1,x+dist,y);\ [s_3,s_4]\ \underline{\text{monitor-single}}(av_1,x+dist,y,x,y)) \mid\mid \\
\quad ([u_1,u_2]\ \textbf{fly-to}(av_2,x-dist,y);\ [u_3,u_4]\ \underline{\text{monitor-single}}(av_2,x-dist,y,x,y)) \mid\mid \\
\quad ([v_1,v_2]\ \textbf{fly-to}(av_3,x,y+dist);\ [v_3,v_4]\ \underline{\text{monitor-single}}(av_3,x,y+dist,x,y)) \mid\mid \\
\quad ([w_1,w_2]\ \textbf{fly-to}(av_4,x,y-dist);\ [w_3,w_4]\ \underline{\text{monitor-single}}(av_4,x,y-dist,x,y))) \\
\text{where} \\
\quad s_3=u_3=v_3=w_3 \wedge s_4=u_4=v_4=w_4 \wedge\ s_4-s_3 \geq \text{minduration}
\end{array}
$$

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

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

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

As shown below, each AV involved in this task iterates while there remains at least one cell (x,y) that it has been assigned ("owns") and that is not yet scanned. In each iteration the variables (x',y') declared in the nested with clause range over arbitrary coordinates, but the associated where clause ensures that only coordinates that belong to the given AV and that have not already been scanned can be selected. Also in each iteration, t_c is bound to the time at which the constraint condition is tested and u,u' are bound to the timepoints at which the inner composite action is performed. The repeated use of u,u' is intentional: The **scan-cell** action will occur over exactly the same interval as the enclosing composite action construct.

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

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

$$[0,1000]([0,1000]\ \underline{\text{scan-with-all-uavs}}() \mid\mid [0,300]\ \underline{\text{monitor-pattern}}(20,25,3))$$

It should be emphasized that in the expected case, the task of generating specifications of this kind would be aided by libraries of predefined domain-related actions as well as by user interfaces adapted to the task at hand. The structure and high-level nature of the language remains important when ensuring that these tools and their output are both correct and comprehensible to a human operator inspecting a mission definition.

10 Collaborative Systems

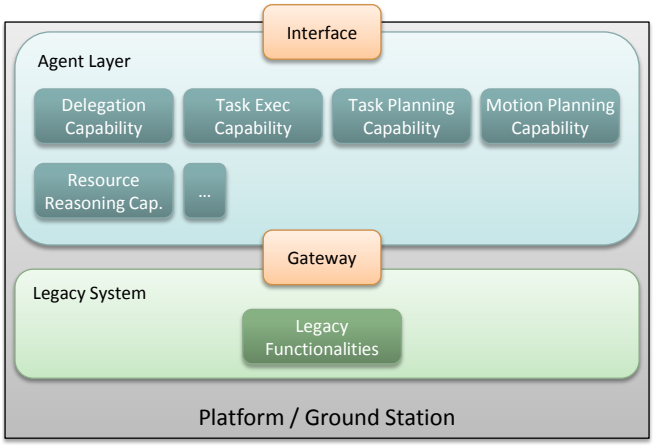
Though the main focus of this chapter has been on the operation of a single unmanned aircraft, issues related to cooperation and collaboration are also essential for the successful use of such systems.

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

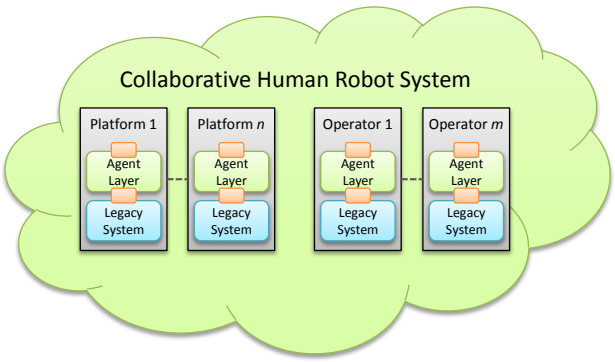
When an unmanned system is viewed as an agent acting on behalf of humans, it is also natural to view the assignment of a complex mission to that system as *delegation* – by definition, the act of assigning authority and responsibility to another person, or in a wider sense an agent, in order to carry out specific activities. Delegation is therefore part of the foundation for collaboration in this architecture: A *delegator* can ask a *contractor* to take the responsibility for a specific *task* to be performed under a set of *constraints*. Informally, an agent receiving a delegation request must verify that to the best of its knowledge, it will be able to perform the associated task under the given constraints, which may for example concern resource usage or temporal and spatial aspects of the mission. To ensure that a complex task is carried out in its entirety, an agent may have to enlist the aid of others, which can then be delegated particular parts of the task. This results in a network of responsibilities between the agents involved and can continue down to the delegation of elementary, indivisible actions. If such a network can be generated in a way that satisfies the associated constraints, the contractor can accept the delegation and is then committed to doing everything in its power to ensure the task is carried out. If not, it must refuse. See Doherty et al. (2013) for a formal characterization of delegation in terms of speech acts as well as a practical delegation protocol that also determines how to allocate tasks to specific agents.

Delegation requires a flexible task structure with a clear formal semantics. This role is played by Task Specification Trees (Section 6.3), whose hierarchical nature also leads to a natural recursive decomposition of tasks where the children of a node are the subtasks that can be re-delegated to other agents. For example, an automated multi-agent planner (Section 9.2) can generate a TST where elementary action nodes are not assigned to specific agents. Then, the delegation process and its embedded *task allocation* functionality (Doherty et al., 2013) can recursively determine how the plan can be executed in a way that satisfies associated constraints. The formal semantics of the task being delegated is specified through a close connection to TALF (Section 9.4). See Doherty et al. (2013, 2011); Doherty and Meyer (2012) for further details about delegation, its close relation to *adjustable autonomy* and *mixed-initiative interaction*, and its integration with automated planning.

Legacy Systems. When an agent-based architecture is used together with an existing platform such as an unmanned aircraft, there may already be an existing legacy system providing a variety of lower-level functionalities such as platform-specific realizations of elementary tasks and resources. Existing interfaces to such functionalities can vary widely. The current instantiation of the architecture (Figure 30(a)) directly supports the use of such legacy functionalities through the use of an agent layer and a gateway. The *agent layer* (Figure 30(b)) encapsulates higher-level deliberative functionalities and provides a common interface for multi-agent collaboration in complex missions, including support for mission specification languages,



(a) Agentified platform or ground control station.



(b) Overview of the collaborative human/robot system.

Figure 30: Agents in a collaborative system.

delegation, and planning. The *gateway* must have a platform-specific implementation, but provides a common platform-independent external interface to the available legacy functionalities. In essence, this allows newly developed higher-level functionalities to be seamlessly integrated with existing systems, without the need to modify either the agent layer or the existing system. The agent layer can then be developed independently of the platforms being used.

Legacy control stations and user interfaces that human operators use to interact with robotic systems are treated similarly, through the addition of an agent layer. The result is a collaborative human/robot system consisting of a number of human operators and robotic platforms each having an agent layer and possibly a legacy system, as shown in Figure 30(b).

11 Mission Applications

The research methodology used during the development of the HDRC3 architecture 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 AVs and deploying them for empirical testing at an early stage. One then iterates and continually increases the robustness and functionalities of the targeted components.

Due to the architecture described in the previous sections it is relatively easy to build on top of existing functionalities and to add new ones in order to put together sophisticated autonomous missions. Below, two such example mission applications are described.

11.1 Emergency Services Assistance

The first application focuses again on the ambitious emergency services scenario discussed in the introduction. An emergency relief scenario in this context can be divided into two separate legs or parts.

11.1.1 Mission Leg I: Body Identification

In the first leg of the mission, a large region should be cooperatively scanned with one or more AVs to identify injured civilians. The result of this scan is a *saliency map* pinpointing potential victims, their locations and associated information such as high resolution photos and thermal images. This information could then be used directly by emergency services or passed on to other AVs as a basis for additional tasks.

Algorithm 4 Saliency map construction

```
1:  Initialize data structures
2:  while scanning not finished do
3:      Simultaneously grab two images: img_color, img_thermal
4:      Analyze img_thermal to find potential human body regions
5:      for each region in img_thermal do
6:          Find corresponding region r_color in img_color
7:          Compute geographical location loc of r_color
8:          Execute human body classifier on r_color
9:          if classification positive then
10:             if loc is new then
11:                 add location loc to map, initialize certainty factor p_body(loc)
12:             else
13:                 update certainty factor p_body(loc)
14:             end if
15:         end if
16:     end for
17: end while
```

A multi-platform area coverage path planning algorithm computes paths for n heterogeneous platforms guaranteeing complete camera coverage, taking into account sensor properties and platform capabilities.

Two video sources (thermal and color) are used, allowing 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 an AV in order not to miss potential victims.

A thermal image is first analyzed to find human body sized silhouettes. The 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 body classification at a rate up to 25 Hz. This high processing rate allows for collecting statistics about classified humans and pruning 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 UASTL RMAX helicopter platform and is an important component in the lab's research with autonomous search and rescue missions.

Saliency Map Construction. Information obtained from thermal and color video streams must be fused in order to create saliency maps of human bodies. An overview of the method used is presented in Algorithm 4. The execution of this algorithm starts when the host AV arrives at the starting position of the area to scan and is terminated when the scanning flight is finished. Its output is a set of geographical locations loc_i and certainty factors $p_{body}(loc_i)$. Refer to (Doherty and Rudol, 2007; Rudol and Doherty, 2008) for additional details.

After initializing the necessary data structures (line 1) the algorithm enters the main loop (line 2), which is terminated when the entire area has been scanned. The main loop begins with simultaneously grabbing two video frames. The thermal image is analyzed first (line 4) to find a set of regions of intensities which correspond to human body temperatures (details below). Then (line 6), for each of these subregions a correspondence in the color frame, as well its geographical location loc, are calculated (details below). The calculated corresponding region in the color frame is analyzed with a human body classifier to verify the hypothesis that the location loc contains a human body (details below). If the classification is positive and the location loc has not been previously identified, then loc is added to the map and its certainty factor initialized (line 11). Otherwise, the certainty factor of that location is updated (line 13, details below).

Thermal Image Processing. The image processing algorithm takes a pair of images as input and starts by analyzing the thermal image (top row of Figure 31). The image is first thresholded to find regions of certain intensities which correspond to human body temperature. The image intensity corresponding to a certain

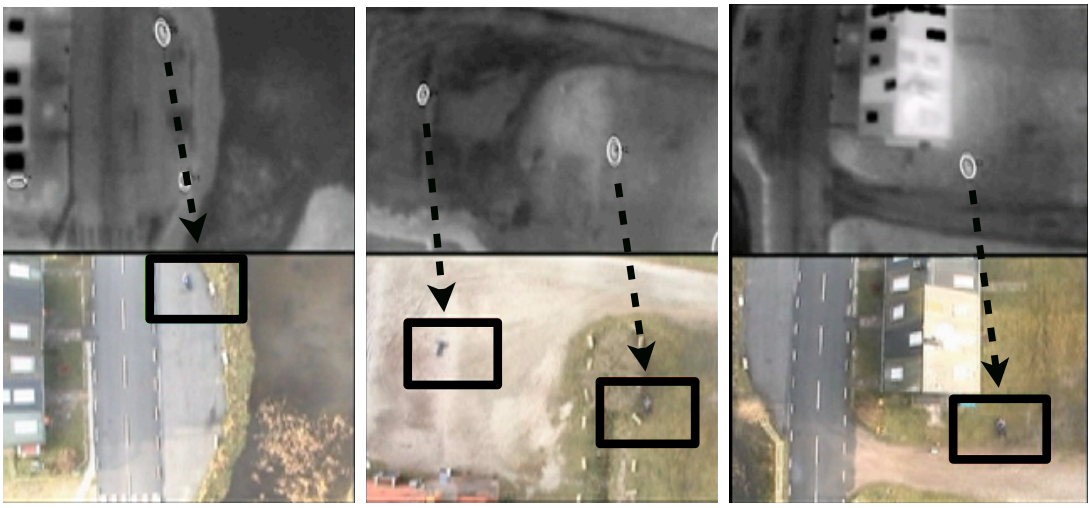


Figure 31: Example input to the image processing algorithm: Thermal images and corresponding color images.

temperature is usually given by the camera manufacturer or can be calibrated by the user. The shapes of the thresholded regions are analyzed and those which do not resemble a human body, due to the wrong ratio between minor and major axes of the fitted ellipse or due to incorrect sizes, are rejected. Once human body candidates are found in the thermal image, corresponding regions in the color image are calculated.

Image Correspondences and Geolocation. Finding corresponding regions using image registration or feature matching techniques is infeasible because of the different appearance of features in color and thermal images. Therefore a closed form solution, which takes into account information about the cameras' pose in the world, is preferred. In short, a geographical location corresponding to pixel coordinates for one of the cameras is calculated. It takes into account the camera's extrinsic and intrinsic parameters and assumes a flat world. The obtained geographical location is then projected back into the other camera image.

The method can be extended to relax the flat world assumption given the elevation model. A geographical location of a target can be found by performing ray-tracing along the line going from the camera center through a pixel to find the intersection with the ground elevation map.

The accuracy of the correspondence calculation is influenced by several factors. All inaccuracies of parameters involved in calculating the position and attitude of cameras in the world contribute to the overall precision of the solution. The evaluation of the accuracy involves investigating the AV state estimation errors, pan-tilt unit position accuracy, camera calibration errors etc. In the case of the UASTL RMAX, during the experimental validation, the corresponding image coordinates were within a subregion of 20% of the video frame size (see the marked subregions in the color images in Figure 31).

Human Body Classification. After calculating the coordinates of the pixel in the color image, a region with the P_c as center (the black rectangles in the bottom row images of Figure 31) is analyzed by an object classifier. The classifier used was first suggested in Viola and Jones (2001). It uses a cascade of classifiers for object detection, and includes a novel image representation, the integral image, for quick detection of features. Classifiers in the initial stages remove a large number of negative examples. Successive classifiers process a smaller number of sub-windows. The initial set of sub-windows can include all possible sub-windows or can be the result of a previous classification in a thermal image additionally improving processing rate. The method was also improved, for example in Lienhart and Maydt (2002) by extending the original feature set.

The classifier requires training with positive and negative examples. During the learning process the structure of a classifier is built using boosting. The use of a cascade of classifiers allows for dramatic speed up of computations by skipping negative instances and only computing features with high probability for

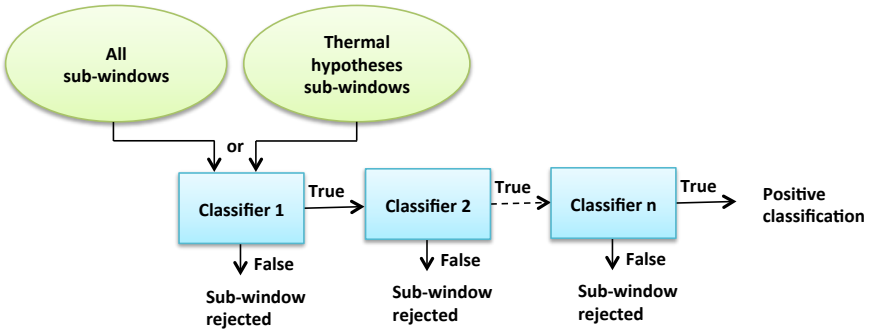


Figure 32: Schematic description of a cascade classifier.

positive classification. The speed up comes from the fact that the classifier, as it slides a window at all scales, works in stages and is applied to a region of interest until at some stage the candidate is rejected or all the stages are passed (see Figure 32). This way, the classifier quickly rejects subregions that most probably do not include the features needed for positive classification (that is, background processing is quickly terminated).

The implementation of the classifier used in this work is a part of the Open Source Computer Vision Library (`http://opencv.org/`) and the trained classifier for upper, lower and full human body is a result of Kruppa et al. (2003). The classifier is best suited for pedestrian detection in frontal and backside views which is exactly the type of views an AV has when flying above the bodies lying on the ground.

The classifier parameters have been adjusted to minimize false negatives: In case of rescue operations it is better to find more false positives than to miss a potential victim. The number of neighboring rectangles needed for successful identification has been set to 1 which makes the classifier accept very weak classifications. The search window is scaled up by 20% between subsequent scans.

Map Building. Since the body classifier is configured to be "relaxed" it delivers sporadic false positive classifications. The results are pruned as follows. Every salient point in the map has two parameters which are used to calculate the certainty of a location being a human body: T_{frame} which describes the amount of time a certain location was in the camera view and T_{body} which describes the amount of time a certain location was classified as a human body. The certainty factor is calculated by $p_{body}(loc_i) = \frac{T_{body}}{T_{frame}}$. A location is considered a body if $p_{body}(loc_i)$ is larger than a certain threshold (0.5 was used during the flight tests) and T_{frame} is larger than a desired minimal observation time. Locations are considered equal if geographical distance between them is smaller than a certain threshold (depending on the geolocation accuracy) and the final value of a geolocated position is an average of the observations.

Experimental Validation. The presented technique for geolocation and saliency map building has been integrated as a perception functionality of the Control Kernel (Section 5.2) and tested in flight. A mission has then been constructed in the shape of a Task Specification Tree that indirectly makes use of the take-off, hovering, path following, and landing flight control modes (through FCL commands, Section 6.2.2), as well as the pan-tilt unit (through PPCL commands, Section 6.2.3).

Test flights were then performed at the Swedish Rescue Services Agency Test Area (Figure 33A). Since this is a city-like closed area with road structures, buildings etc., the video streams included different types of textures such as grass, asphalt, gravel, water and building rooftops. An example complete push button mission setup was as follows:

- Two UASTL RMAX helicopters were used starting from H_1 and H_2 in Figure 33B.
- An operator selected a rectangular area on a map where the saliency map was to be built (Figure 33B).
- On-board systems calculated the mission plan taking into account properties of the on-board sensors (such as the field of view) of both AVs. The plan consisted of two separate flight plans for the two

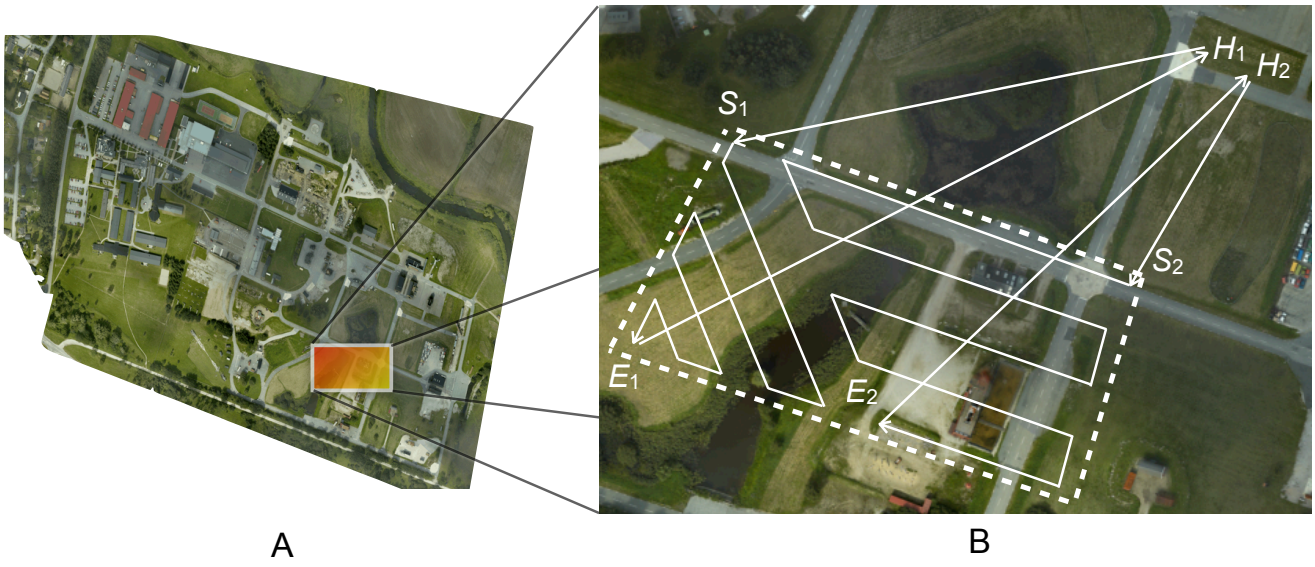


Figure 33: A. Map of the Swedish Rescue Services Agency Test Area in Revinge, B. A closeup view of the area where the saliency map was built. Approximate flight paths are marked with solid lines.

AVs.

- The mission started with simultaneous takeoffs and flying to starting positions S_1 and S_2 in Figure 33B. After arriving at the starting positions the execution of the scanning paths autonomously began and the saliency map building was initiated.
- Upon finishing the scanning paths at positions E_1 and E_2, the AVs flew to the takeoff positions and performed autonomous landings.

Experimental Results. All eleven human bodies placed in the area were found and geolocated. Corresponding color and thermal images for the identified objects are displayed vertically as pairs in Figure 34. Images 7, 9 and 14 present three falsely identified objects, caused by configuring the human body classifier to accept weak classifications. A more restrictive setup could instead result in missing potential victims. The images could additionally be judged by a human operator to filter out the false-positive classifications. Both human bodies and dummies were detected despite the lower temperature of the latter.

Figure 35 presents the generated saliency map. The scanning pattern segments for the platform starting in position H_2 is marked with a solid line, and its final position is marked with a cross icon. The fields of view of the color and thermal cameras are depicted with light-grey and black rectangles, respectively. These differ slightly as the two cameras have different properties as identified during calibration procedures.

The circles indicate the identified body positions. The lighter the shade of the color, the more certain the classification. As can be seen the most certain body positions are objects number 2 and 11 (in Figure 34). It is due to the fact that these body images are clearly distinguishable from the homogeneous background. Nevertheless, even body images with more cluttered backgrounds were identified.

The accuracy of the body geolocation calculation was estimated by measuring GPS positions (without differential correction) of bodies after an experimental flight. The measurement has a bias of approximately two meters in both east and north directions. It is the sum of errors in GPS measurement, accuracy of the camera platform mounting, PTU measurement and camera calibration inaccuracies. The spread of measurement samples of approximately 2.5 meters in both east and north directions is a sum of errors of the AV's attitude measurement, the system of springs in the camera platform and time differences between the AV state estimate, PTU angle measurement and image processing result acquisition.

The presented algorithm requires only a single pair of images for human body classification. In practice,

Figure 34: Images of classified bodies. Corresponding thermal images are placed under color images.

however, the more pairs available the more certain the result of a classification can become. Additionally, thanks to using the results of the thermal image analysis to focus the classification in the color image subregions, a high rate of processing is achieved (above 20 Hz for the presented results).

11.1.2 Mission Leg II: Package Delivery

After successful completion of leg I of the mission scenario, one 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 AVs 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.

For these logistics missions, the use of one or more AVs with diverse roles and capabilities is assumed. Initially, there are n injured body locations, several supply depots and several supply carrier depots (see Figure 37). The logistics mission is comprised of one or more AVs transporting boxes containing food and medical supplies between different locations (Figure 36). Plans are generated in the millisecond to seconds range using TALplanner (see Section 9.2) and empirical testing shows that this approach is promising in terms of integrating high-level deliberative capability with lower-level reactive and control functionality.

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 AV has a device for attaching to boxes and carrying

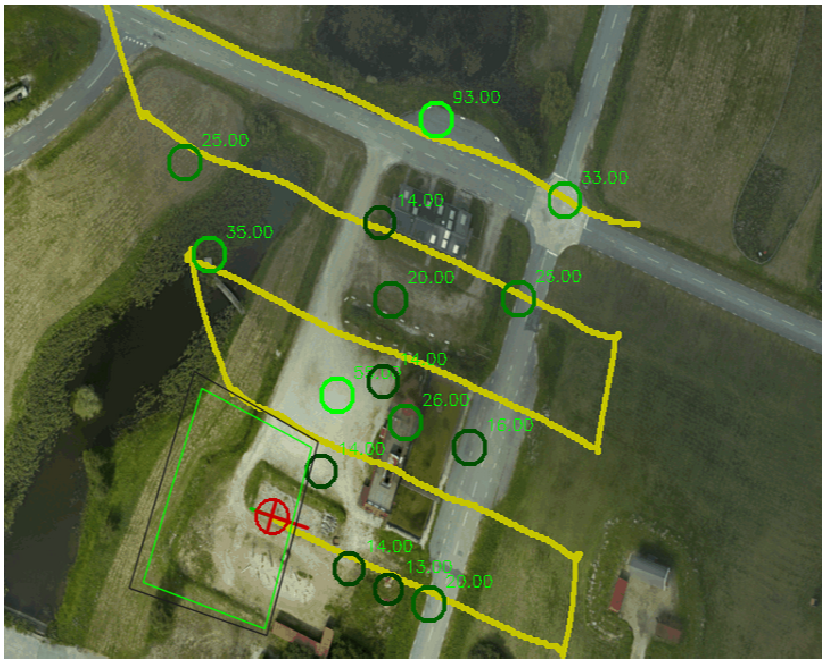


Figure 35: The resulting map with salient points marked as circles. The lighter the shade of the color the higher the detection certainty.

them beneath the AV. 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 an AV increases manifold over longer distances. The ultimate mission for the AVs is to transport the food and medical supplies to their destinations as efficiently as possible using the carriers and boxes at their disposal.

A physical prototype of a mechanism for carrying and releasing packages has been developed and tested. Figure 38 presents two images of the prototype system. The logistics scenario has also been tested in a simulated AV 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 control software as the physical AV. The execution monitor system has been tested in this simulation as well, with a large variety of deviations tested through fault injection in the simulation system. The simulator makes use of the Open Dynamics Engine (`http://www.ode.org`), 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.

11.2 Map Building using a Laser Range Finder

The second application presented here deals with the construction of an elevation map using a laser range finder allowing AVs to navigate safely in unknown outdoor environments.

As described in Section 7.1, the path planning algorithms used here are based on a geometrical description of the environment to generate collision-free paths. The safety of a UAS operation therefore depends on having a sufficiently accurate 3D model of the environment. Predefined maps may become inaccurate or outdated over time because of the environment changes, for example due to new building structures and vegetation growth. Therefore adequate sensors and techniques for updating or acquiring new 3D models of the environment are necessary in many cases.

Figure 36: The AV logistics simulator

Figure 37: A supply depot (left) and a carrier depot (right)

Among the many sensors available for providing 3D information about an operational environment, laser range finders provide high accuracy data at a reasonable weight and power consumption. One of the reasons for the innovation in this particular sensor technology is its wide use in many industries, but laser range finders have also received a great deal of interest from the robotics community, where their main usage is in navigation and mapping tasks for ground robotic systems, such as localization (Burgard et al., 1997), 2D Simultaneous Localization and Mapping (SLAM (Montemerlo and Thrun, 2007)), 3D SLAM (includes 3D position (Cole and Newman, 2006)), and 6D SLAM (includes 3D position and attitude (Nüchter et al., 2004)).

The device integrated with the UASTL RMAX system is the popular LMS-291 from SICK AG (`http://www.sick.com`). This unit does not require any reflectors or markers on the targets nor scene illumination to provide real-time measurements. It performs very well both in indoor and outdoor environments. The system is equipped with a rotating mirror which allows for obtaining distance information in one plane in front of the sensor with a selectable field of view of 100 or 180 degrees (Figure 39). It gives a resolution of 1 cm with a maximum range of 80 meters, or a resolution of 1 mm with a range of 8 meters, an angular resolution of 0.25, 0.5 or 1.0 degrees, and a corresponding response time of 53, 26 or 13 ms.

The laser unit has been modified to reduce its weight from 4.5 to 1.8 kg. It has then been mounted on an in-house developed rotation mechanism supporting continuous rotation of the sensor around the middle

Figure 38: Two frames of video presenting the prototype mechanism for carrying and releasing packages using an electromagnet. An arrow points to a package being carried. The top left picture presents the on-board camera view.

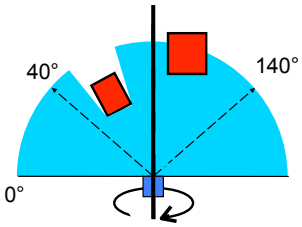


Figure 39: Top view of the LMS-291 scanning field and the axis of rotation using the rotation mechanism.

laser beam (solid line in Figure 39), which allows for obtaining half-sphere 3D point clouds even when the vehicle is stationary. A similar approach to the integration of the laser range finder with an UASTL RMAX platform is used by Whalley et al. (2008).

System Integration. The 3D map building algorithm based on the data provided by LRF sensor has been integrated as a perception functionality of the Control Kernel (Section 5.2) and tested in flight. A Task Specification Tree node has been specified to achieve the required mission goals (Section 6.3). Similar to the previous application example, the take-off, hovering, path following, and landing flight control modes were used during the mission through Flight Control Language commands (Section 6.2.2). Additionally, the Payload and Perception Control Language (Section 6.2.3) are used to set a constant angular speed of the LRF rotational mechanism as well as the distance and angular resolutions of the sensor. The 3D maps acquired are saved in the GIS Service database and available to the deliberative services (such as path planners, discussed in Section 7).

Experimental Results. Several missions were performed during which both the LRF data and AV state estimates were collected and integrated. Figure 40 presents a reconstructed elevation map of the Revinge flight test area (the left side) focusing on two building structures. A photo of corresponding buildings is presented on the right side of the figure. The elevation map is built by sampling the LRF data with 1 meter resolution and constructing a set of triangles in order to represent the elevation.

In order to assess the fidelity of the newly generated model an overlay with the existing model was generated. The result is presented in Figure 41a. The new map includes some changes in the environment, including a metal container on the left in the figure (A.) and new vegetation (B.) that was not present in the existing model.

The new models stored by the GIS Service can be used by the AV platform for path planning in order to generate collision-free paths (see Section 7.1). Since generated models are based on noisy measurements a

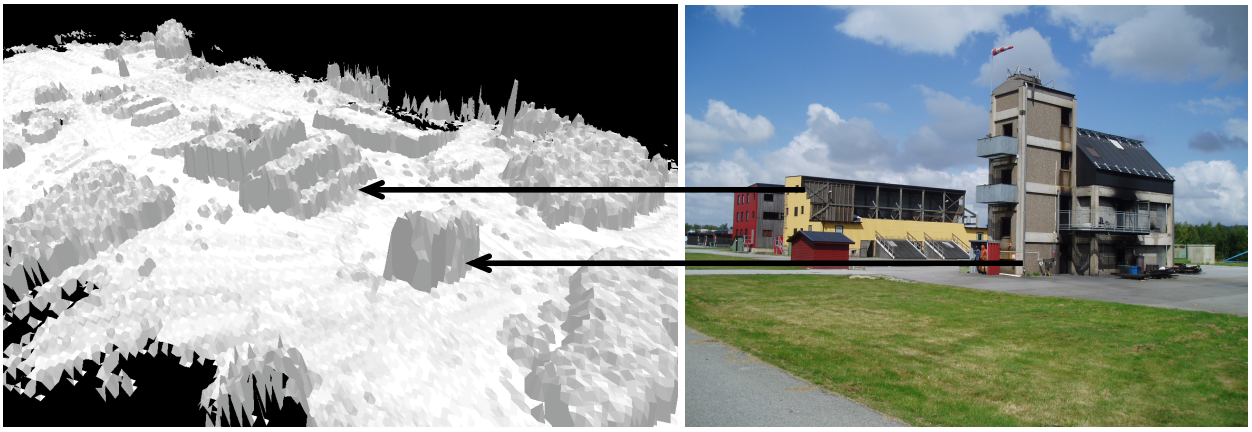
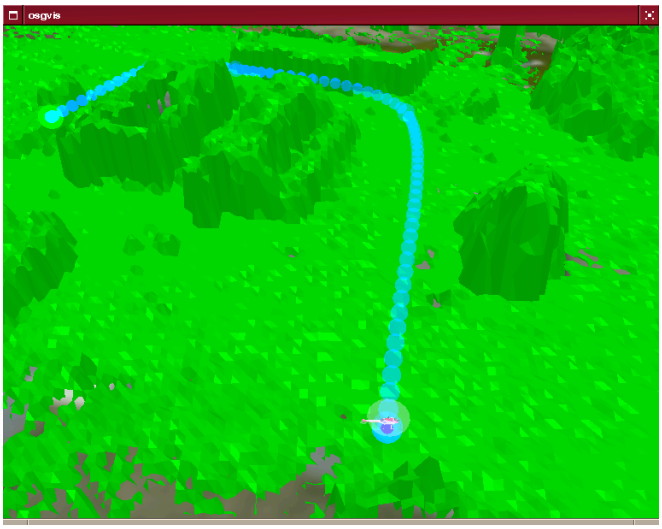


Figure 40: Overview of the reconstructed elevation map of the Revinge flight test area based on the laser range finder data (left) and a photo of corresponding building structures (right).

(a) Overlay of the new elevation map with the existing Revinge flight test area model.



(b) Example of path planner use in the reconstructed map of the Revinge area.

Figure 41: Results of the map building procedure.

safety margin during the planning is used. For the UASTL RMAX, a safety margin of 6 meters is used. An example of path generated by the path planner using the new model is presented in Figure 41b.

The accuracy of models built with the raw LRF point clouds (without applying the scan matching algorithms) is sufficient for navigation purposes if the necessary safety margins are used. The inaccuracies introduced by the measurement errors and the uncertainty of the AV state estimate might result in narrowing down the operational environment of the AV. For example, in Figure 41a a narrow passage (C.) can be excluded from the collision-free space although the corridor between the two buildings is wide enough to fly through. Further investigation of methods for improving the model quality is ongoing at the time of writing this chapter.

12 Conclusions

The goal of this research is the development of autonomous intelligent systems for unmanned aircraft. This chapter has described a hybrid deliberative reactive architecture which has been developed through the years and successfully deployed on a number of unmanned aircraft. The focus was on an instantiation of

this architecture with the UASTL RMAX. The architecture was described in a manner that should be useful to the unmanned aircraft research community, because it isolates generic functionality and architectural solutions that can be transferred and used on existing unmanned aircraft and also those envisioned for the future. The experimental development of such systems has wider impact since the HDRC3 architecture and its instantiation on the UASTL RMAX is an example of a new type of software system commonly known as a *software intensive system* or *cyber physical system*. Such systems are characterized by a number of features: the complex combination of both hardware and software; the high degree of concurrency used; the distributed nature of the software; the network-centric nature of the environment in which they reside; the open nature of these systems; and finally, the requirement that they are both scalable and adaptive since specification of such systems is necessarily open due to their interaction with other components not part of the system. The latter features are particularly important in the context of collaborative missions with other heterogeneous systems.

The HDRC3 architecture encapsulates many of these features. Emphasis has been placed on the importance of clean transitional interfaces between the different layers of the architecture which are inhabited by computational processes with diverse requirements in terms of temporal latency in the decision cycles and the degree to which the processes use internal models. The HDRC3 architecture is also highly modular and extensible.

The use of HCSMs separates specification of the behavior of the system from the atomic conditions and activities which are implemented procedurally. This permits the use of a real-time interpreter for HCSMs and alleviates the requirement for recompilation when new HCSMs are added to the system. The approach is also amenable to distribution where federations of HCSM interpreters can inter-communicate with each other on a single platform as is the case with the UASTL RMAX system, but also across platforms if required.

The Platform Server provides a clean declarative interface to both the suite of flight control modes and perception control modes through the use of two languages, FCL and PPCL, respectively. This approach provides a primitive language of building blocks that can be structured and used by other functionalities in the system to define higher level task specifications. Extending the suite of basic actions in the UASTL RMAX system is done by defining new HCSMs which interface to new or existing control modes and then extending the language interface in the Platform Server.

The task specifications themselves have a well-defined and extensible language, Task Specification Trees, which are used by both the reactive and deliberative layers of the architecture as a declarative specification language for tasks. Primitive actions defined through the Platform Server interface can be used with TSTs or if required, new actions can be defined by specifying new node types. The execution and procedural implementation of these basic action and new node types is separated from their declarative specification through the use of node executors and a node execution facility in the reactive layer of the HDRC3 architecture.

This clean separation has a number of advantages besides extensibility and modularity. A declarative specification of tasks permits the sharing of tasks implemented in different ways across platforms. This is especially important in the context of collaborative robotics and heterogeneous robotic systems. It also permits the translation of other task languages into a common representation. The translation of the output of automated task planners is a case in point. There is a direct mapping from the declarative output of a planner to grounded executable components in the architecture.

The DyKnow system is another extensible and modular functionality. Any source in an architecture can become a stream-based data source. One naturally thinks of sensors as specific sources of data, but at another extreme one can think of another platform as a source of streamed data, or a data base or the Internet. The DyKnow system provides means for specifying, constructing, generating, using and managing these diverse sources of data contextually at many different levels of abstraction. This is a unique component in the HDRC3 architecture that has had widespread use by other functionalities such as the execution monitoring system and the task and motion planners.

Traditionally, verification and validation of autonomous systems has been a central research issue. In the context of unmanned aircraft, a great deal of research has focused on the control kernel and to some extent the lower parts of the reactive layer. In terms of the reactive layer itself and in particular the highly non-deterministic functionality associated with the deliberative layer, very little effort has been put into verification and validation because tools simply do not exist yet. A great deal of research effort has gone into improving this absence of effort during the development of key functionalities in the HDRC3 architecture. Many of the functionalities there such as the task planner, the task specification language, the mission specification language and the execution monitoring system, each have a formal semantics based on the use of Temporal Action Logics.

In fact the use of logic is highly integrated with many of the processes evoked by deliberative functionality. The execution monitoring system is essentially a real-time dynamic model-checking system where temporal models are generated dynamically by DyKnow during the course of operation and various constraints are checked relative to those models by evaluating formulas through progression and checking for their satisfiability. The output of TALplanner is a formal narrative structure in Temporal Action Logic. Consequently both the process of generating a plan and the resulting output can be reasoned about using inference mechanisms associated with TALplanner.

In summary, this chapter has described an empirically tested unmanned aircraft architecture that combines many years of both engineering and scientific insight acquired through the successful deployment of the UASTL RMAX system in highly complex autonomous missions. To the extent possible, these insights have been described in a generic manner in the hope that many of the functionalities described can serve as a basis for use by others in the continued and exciting development of such systems.

Acknowledgments

This work is partially supported by the EU FP7 project SHERPA (grant agreement 600958), the Swedish Foundation for Strategic Research (SSF) Collaborative Unmanned Aircraft Systems (CUAS) project, the Swedish Research Council (VR) Linnaeus Center for Control, Autonomy, Decision-making in Complex Systems (CADICS), and the ELLIIT network organization for Information and Communication Technology.

References

J. S. Albus, F. G. Proctor. A reference model architecture for intelligent hybrid control systems. *Proceedings of the International Federation of Automatic Control (IFAC)*, pp. 1–7, 2000

R. C. Arkin. *Behavior-based robotics* (The MIT Press, 1998)

F. Bacchus, F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22, 1998

K. Ben Lamine, F. Kabanza. 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-Verlag, 2002)

G. Berry. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19:87–152, 1992. doi:10.1016/0167-6423(92)90005-V

G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide* (Addison-Wesley Professional, 2005). ISBN 978-0321267979

A. Brooks, T. Kaupp, A. Makarenko, S. Williams, A. Orebäck. Towards component-based robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2005)

R. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 692–696 (1989)

W. Burgard, D. Fox, S. Thrun. Active mobile robot localization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (Morgan Kaufmann, 1997)

D. Cole, P. Newman. Using laser range data for 3D SLAM in outdoor environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (2006)

G. Conte. *Navigation Functionalities for an Autonomous UAV Helicopter*. Licentiate thesis 1307, Department of Computer and Information Science, Linköping University, 2007

G. Conte. *Vision-Based Localization and Guidance for Unmanned Aerial Vehicles*. Ph.D. thesis, Department of Computer and Information Science, Linköping University, Linköping, 2009

G. Conte, P. Doherty. Vision-based unmanned aerial vehicle navigation using geo-referenced information. *EURASIP Journal of Advances in Signal Processing*, 2009

G. Conte, S. Duranti, T. Merz. Dynamic 3D path following for an autonomous helicopter. In *Proceedings of the IFAC Symposium on Intelligent Autonomous Vehicles* (2004)

J. Cremer, J. Kearney, Y. Papelis. HCSM: a framework for behavior and scenario control in virtual environments. *ACM Transactions on Modeling and Computer Simulation*, 1995

G. De Giacomo, R. Reiter, M. Soutchanski. Execution monitoring of high-level robot programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 453–465 (Morgan Kaufmann, 1998)

P. Doherty, J. Gustafsson, L. Karlsson, J. Kvarnström. (TAL) temporal action logics: Language specification and tutorial. *Electronic Transactions on Artifical Intelligence*, 2(3-4):273–306, 1998. ISSN 1403-3534

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

P. Doherty, F. Heintz, J. Kvarnström. High-level Mission Specification and Planning for Collaborative Unmanned Aircraft Systems using Delegation. *Unmanned Systems*, 2013. ISSN 2301-3850, EISSN 2301-3869. Accepted for publication.

P. Doherty, F. Heintz, D. Landén. A delegation-based architecture for collaborative robotics. In *Agent-Oriented Software Engineering XI: Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 6788, pp. 205–247 (Springer-Verlag, 2011)

P. Doherty, J. Kvarnström. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *6th International Workshop on Temporal Representation and Reasoning (TIME-99)*, pp. 47– (IEEE Computer Society, 1999). ISBN 0-7695-0173-7

P. Doherty, J. Kvarnström. TALplanner: A temporal logic-based planner. *Artificial Intelligence Magazine*, 22(3):95–102, 2001

P. Doherty, J. Kvarnström. Temporal action logics. In *The Handbook of Knowledge Representation*, chapter 18, pp. 709–757 (Elsevier, 2008)

P. Doherty, J. Kvarnström, F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009

P. Doherty, J. Kvarnström, A. Szalas. Temporal composite actions with constraints. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 478–488 (2012)

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

P. Doherty, J.-J. C. Meyer. On the logic of delegation: Relating theory and practice. In *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi* (College Publications, London, 2012)

P. Doherty, P. Rudol. A UAV search and rescue scenario with human body detection and geolocalization. In *AI'07: Proceedings of the Australian joint conference on Advances in artificial intelligence*, pp. 1–13 (Springer-Verlag, 2007). ISBN 3-540-76926-9, 978-3-540-76926-2

S. Duranti, G. Conte. In-flight identification of the augmented flight dynamics of the RMAX unmanned helicopter. *Proceedings of the IFAC Symposium on Automatic Control in Aerospace*, 2007

M. Egerstedt, X. Hu, A. Stotsky. Control of mobile platforms using a virtual vehicle approach. *IEEE Transactions on Automatic Control*, 46(11):1777–1782, 2001

J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong. Taming heterogeneity – the Ptolemy approach. In *Proceedings of the IEEE*, pp. 127–144 (2003)

E. A. Emerson. *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*, chapter Temporal and Modal Logic, pp. 997–1072 (Elsevier and MIT Press, 1990)

J. L. Fernández, R. G. Simmons. Robust execution monitoring for navigation plans. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (1998)

M. Fichtner, A. Grossmann, M. Thielscher. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57(2-4):371–392, 2003. ISSN 0169-2968

R. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial intelligence*, volume 1, pp. 202–206 (Seattle, WA, 1987)

Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. http://www.fipa.org, 2002

E. Gat. On three-layer architectures. *Artificial intelligence and mobile robots*, 1997

E. Gat, M. G. Slack, D. P. Miller, R. J. Firby. Path planning and execution monitoring for a planetary rover. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 20–25 (IEEE Computer Society Press, 1990)

M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 597–607 (1996). ISBN 1-55860-421-9

S. Gottschalk, M. C. Lin, D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the Annual Conference on Computer graphics and interactive techniques* (1996)

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987

F. Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework*. Ph.D. thesis, Department of Computer and Information Science, Linköping University, 2009

F. Heintz, P. Doherty. DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems*, 15(1):3–13, 2004

F. Heintz, P. Doherty. 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, 2006

F. Heintz, Z. Dragisic. Semantic information integration for stream reasoning. In *Proceedings of the International Conference on Information Fusion* (2012)

F. Heintz, J. Kvarnström, P. Doherty. Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, 2010

F. Heintz, J. Kvarnström, P. Doherty. Stream-based hierarchical anchoring. *Künstliche Intelligenz*, 27:119–128, 2013. doi:10.1007/s13218-013-0239-2

F. Heintz, P. Rudol, P. Doherty. From images to traffic behavior – a UAV tracking and monitoring application. In *Proceedings of the International Conference on Information Fusion*, pp. 1–8 (2007)

I. Horrocks. Ontologies and the Semantic Web. *Communications of the ACM*, 51(12), 2008. ISSN 00010782. doi:10.1145/1409360.1409377

L. E. Kavraki, P. Švestka, J. Latombe, M. H. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996

M. Kleinehagenbrock, J. Fritsch, G. Sagerer. Supporting advanced interaction capabilities on a mobile robot with a flexible control system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2004)

K. Konolige. Colbert: A language for reactive control in Sapphira. In *KI-97: Advances in Artificial Intelligence*, *Lecture Notes in Computer Science*, volume 1303, pp. 31–52 (Springer, 1997)

K. Konolige, K. Myers, E. Ruspini, A. Saffiotti. The saphira architecture: A design for autonomy. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):215–235, 1997

T. Koo, F. Hoffmann, F. Mann, H. Shim. Hybrid control of an autonomous helicopter. *IFAC Workshop on Motion Control*, 1998

R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990

H. Kruppa, M. Castrillon-Santana, B. Schiele. Fast and robust face finding via local context. In *Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance* (2003)

J. J. Kuffner, S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 995–1001 (2000)

J. Kvarnström. *TALplanner and Other Extensions to Temporal Action Logic*. Ph.D. thesis, Department of Computer and Information Science, Linköping University, 2005

J. Kvarnström. Planning for loosely coupled agents using partial order forward-chaining. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 138–145 (AAAI Press, 2011). ISBN 978-1-57735-503-8, 978-1-57735-504-5

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

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

S. M. LaValle. *Planning Algorithms* (Cambridge University Press, 2004)

R. Lienhart, J. Maydt. An extended set of haar-like features for rapid object detection. In *Proceedings of International Conference on Image Processing*, pp. 900–903 (2002)

P. Mantegazza, E. L. Dozio, S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 72, 2000

G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955

T. Merz. Building a system for autonomous aerial robotics research. In *Proceedings of the IFAC Symposium on Intelligent Autonomous Vehicles (IAV)* (2004)

T. Merz, S. Duranti, G. Conte. Autonomous landing of an unmanned aerial helicopter based on vision and inertial sensing. In *Proceedings of the International Symposium on Experimental Robotics* (2004)

T. Merz, P. Rudol, M. Wzorek. Control system framework for autonomous robots based on extended state machines. *Autonomic and Autonomous Systems, International Conference on*, 0:14, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/ICAS.2006.19

M. Montemerlo, S. Thrun. *FastSLAM: A scalable method for the simultaneous localization and mapping problem in robotics*, *Springer Tracts in Advanced Robotics*, volume 27 (Springer-Verlag, 2007). ISBN 978-3-540-46402-0

E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pp. 129–153 (Princeton University Press, 1956)

A. Nüchter, H. Surmann, K. Lingermann, J. Hertzberg, S. Thrun. 6D SLAM with an application in autonomous mine mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1998–2003 (2004)

P.-O. Pettersson. *Using Randomized Algorithms for Helicopter Path Planning*. Licentiate thesis, Department of Computer and Information Science, Linköping University, 2006

P. O. Pettersson, P. Doherty. Probabilistic roadmap based path planning for an autonomous unmanned helicopter. *Journal of Intelligent & Fuzzy Systems*, 17(4):395–405, 2006

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

P. Rudol, P. Doherty. Human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. In *Proceedings of the IEEE Aerospace Conference* (2008)

S. Sekhavat, P. Svestka, J.-P. Laumond, M. H. Overmars. Multi-level path planning for nonholonomic robots using semi-holonomic subsystems. *International Journal of Robotics Research*, 17:840–857, 1996

M. K. Smith, C. Welty, D. L. McGuinness. OWL Web Ontology Language Guide. 2004

P. Viola, M. J.-C. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the Conference on Computer Vision and Pattern Recognition* (2001)

M. Whalley, G. Schulein, C. Theodore. Design and flight test results for a hemispherical LADAR developed to support unmanned rotorcraft urban operations research. *American Helicopter Society 64th Annual Forum*, 2008

F. White. A model for data fusion. In *Proceedings of the National Symposium for Sensor Fusion* (1988)

M. Wzorek. *Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems*. Licentiate thesis 1509, Department of Computer and Information Science, Linköping University, 2011

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

M. Wzorek, D. Landen, P. Doherty. GSM technology as a communication media for an autonomous unmanned aerial vehicle. In *Proceedings of the Bristol International Conference on UAV Systems* (2006b)

Index