DyKnow: A Framework for Processing Dynamic Knowledge and Object Structures in Autonomous Systems

Fredrik Heintz and Patrick Doherty
Dept. of Computer and Information Science
Linköping University, Sweden
{frehe, patdo}@ida.liu.se

Abstract

Any autonomous system embedded in a dynamic and changing environment must be able to create qualitative knowledge and object structures representing aspects of its environment on the fly from raw or preprocessed sensor data in order to reason qualitatively about the environment. These structures must be managed and made accessible to deliberative and reactive functionalities which are dependent on being situationally aware of the changes in both the robotic agent's embedding and internal environment. DyKnow is a software framework which provides a set of functionalities for contextually accessing, storing, creating and processing such structures. The system is implemented and has been deployed in a deliberative/reactive architecture for an autonomous unmanned aerial vehicle. The architecture itself is distributed and uses RT CORBA as a communications infrastructure. We describe the system and show how it can be used in execution monitoring and chronicle recognition scenarios for UAV applications.

1. Introduction

Research in cognitive robotics is concerned with endowing robots and software agents with higher level cognitive functions that enable them to reason, act and perceive in a goal-directed manner in changing, incompletely known, and unpredictable environments. Research in robotics has traditionally emphasized low-level sensing, sensor processing, control and manipulative tasks. One of the open challenges in cognitive robotics is to integrate techniques from both disciplines and develop architectures which support the seamless integration of low-level sensing and sensor processing with the generation and maintenance of higher level knowledge structures grounded in the sensor data.

Knowledge about the internal and external environments of a roboticagent is often both static and dynamic. This is due to the great amount of background or deep knowledge required by the agent in understanding its world and to the dynamics in the embedding environment where objects of interest are cognized, hypothesized as being of a particular type or types and whose dynamics must be continuously reasoned about in a timely manner. This implies signal-to-symbol transformations at many levels of abstraction with different and varying constraints on real-time processing.

Much of the reasoning involved with dynamic objects and the dynamic knowledge related to such objects involves issues of situation awareness. How can a robotics architecture support the task of getting the right information in the right form to the right functionalities in the architecture at the right time in order to support decision making and goal-directed behavior? Another important aspect of the problem is the fact that this is an on-going process. Data and knowledge about dynamic objects has to be provided continuously and on-the-fly at the rate and in the form most efficient for the receiving cognitive or reactive robotics functionality in a particular context.

Context is important here because the most optimal rates and forms in which a robotic functionality receives data are often task and environmentally dependent. Consequently, autonomous agents must be able to declaratively specify and re-configure the character of the data received. By robotic functionalities, we mean control, reactive and deliberative functionalities ranging from sensor manipulation and navigation to high-level functionalities such as chronicle recognition, trajectory planning, execution monitoring, and hypothesis generation.

In order to make these ideas more precise, we will begin with a scenario from an unmanned aerial vehicle project the authors are involved in which requires many of the capabilities discussed so far. We will then proceed to the details of a deployed software architecture which has been used in actual flights and has many of these capabilities.

Picture the following scenario. An autonomous unmanned aerial vehicle (UAV), in our case, a helicopter, is given a mission to identify and track a vehicle with a particular signature in a region of a small city. The signature is provided in terms of color and size (and possibly 3D shape). Assume that the UAV has a 3D model of the region in addition to information about building structures and the road system. These models can be provided or may have been generated by the UAV itself. Additionally, assume the UAV is equipped with a GPS and INS[1] for navigating purposes and that its main sensor is a camera on a pan/tilt mount.

Let's consider the processing from the bottom up, even though in reality, there will be many feedback loops in the UAV architecture. One way for the UAV to achieve its task would be to initiate a reactive task procedure (parent procedure) which calls the systems image processing module with the vehicle signature as a parameter. The image processing module might then try to identify colored blobs in the region of the right size, shape and color as a first step. These object descriptions would have to be sent to a module in the architecture called the dynamic object repository (DOR) which is responsible for the dynamic management of such objects. Each of these *vision objects* would contain features related to the image processing task such as RGB values with uncertainty bounds, length and width in pixels, position in the image, a sub-image of the object which can be used as a template for tracking, an estimate of velocity, etc.

From the perspective of the UAV, these objects are only cognized to the extent that they are moving colored blobs of interest and the feature data being collected should continue to be collected while tracking those objects perceived to be of interest. What objects are of interest? The parent procedure might identify that or those objects which are of interest based on a similarity measure according to size, color and movement. In order to do this, the DOR would be instructed to create one or more *world objects* and link them to their respective vision objects. At this point the object is cognized at a more qualitative level of abstraction, yet its description in terms of its linkage structure contains both cognitive and pre-cognitive information which must be continuously managed and processed due to the interdependencies of the features at various levels.

A world object could contain additional features such as position in a geographic coordinate system rather than the low-level image coordinate. Generating a geographic coordinate from an image coordinate continuously, called *co-location* is a complex process that involves combining dynamic data about features from several different objects such as the camera object, helicopter object and world objects, together with data from an onboard geographical information system (GIS) module which is also part of the architecture. One would require a computational unit of sorts that takes streamed data as input and outputs a new stream at a higher level of abstraction representing the current geographical coordinate of the object. This co-location process must occur in real-time and continually occur as the world object is tracked. This implies that all features for all dynamic objects linked to the world object in focus have to be continually updated and managed.

At this point, the parent task may want to make a comparison between the geographical coordinate and the position of that coordinate in terms of the road system for the region, information of which is stored in the onboard GIS. This indexing mechanism is important since it allows the UAV to reason qualitatively about its spatial surroundings. Let's assume this is done and after some period of tracking and monitoring the stream of coordinates, the parent procedure decides that this looks like a vehicle that is following the road. *on-road* objects might then be created for each of the world objects that pass the test and linked to their respective world objects. An on-road object could contain more abstract and qualitative features such as position in a road segment which would allow the parent procedure to reason qualitatively about its position in the world relative to the road, other vehicles on the road, and other building structures in the vicinity of the road. At this point, streams of data are being generated and computed for many of the features in the linked object structures at many levels of abstraction as the helicopter tracks the on-road objects.

The parent procedure could now use static knowledge stored in onboard knowledge bases and the GIS together with this dynamic knowledge to hypothesis as to the type of vehicle. The hypothesis would of course be based on the linkage structure for an on-road object and various features at different levels of abstraction. Assume the parent procedure hypothesizes that the on-road object is a car. A *car object* could then be created and linked to the existing linkage structure with additional high-level feature information about the car.

Whether or not the sum of streamed data which makes up the linkage structure represents a particular type of conceptual entity will only ever remain a hypothesis which could very well change, based on changes in the character of the streams of data. Monitors, users of these structures, would have to be set up to observe such changes and alert the parent procedure if the changes become too abnormal relative to some criteria determined by the parent procedure. Abnormality is a concept that is well-suited for being reasoned about at a logical level and the streamed data would have to be put into a form amenable to this type of processing.

How then can an architecture be set up to support the processes described in the UAV scenario above? This is the main topic of this paper and in it we propose a software system called the *DyKnow Framework*.[2]

1 GPS and INS are acronyms for geographic positioning system and inertial navigation system, respectively.

The paper is structured as follows. In section 2, the UAV platform used in the project is briefly described. In section 3, DARA, a Distributed Autonomous Robotics Architecture for UAVs is briefly described. DyKnow is an essential module in this architecture. In section 4, the underlying ontology for dynamic knowledge and object structures is described. In section 5, the basic structure and implementation of the DyKnow Framework is described. In sections 6.2 and 6.3, two deliberative functionalities which use the DyKnow Framework are considered, chronicle recognition and execution monitoring, in addition to the dynamic object repository (DOR).

2. The WITAS UAV Platform

The WITAS[3] Unmanned Aerial Vehicle Project [1] is an ambitious, long-term basic research project whose main objectives are the development of an integrated hardware/software VTOL (Vertical Take-Off and Landing) platform for fully-autonomous missions and its future deployment in applications such as traffic monitoring and surveillance, emergency services assistance, photogrammetry and surveying.

The WITAS Project UAV platform we use is a slightly modified Yamaha RMAX (figure 1). It has a total length of 3.6 m (incl. main rotor), a maximum take-off weight of 95 kg, and is powered by a 21 hp two-stroke engine. Yamaha equipped the radio controlled RMAX with an attitude sensor (YAS) and an attitude control system (YACS).

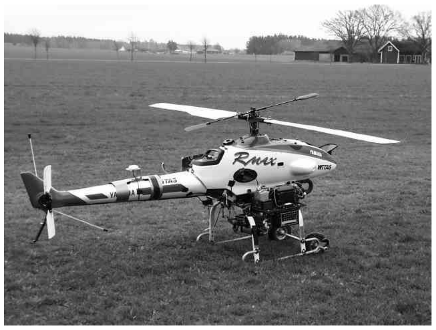

Figure 1. The WITAS RMAX Helicopter

The hardware platform consists of three PC104 embedded computers (figure 2). The primary control system consists of a PIII (700Mhz) processor, a wireless modem (serial line RS232C) and the following sensors: an integrated INS/DGPS (serial), a barometric altitude sensor (analog), a sonar and infrared altimeter (analog), and a compass (serial). It is connected to the YAS and YACS (serial), the image processing computer (serial) and the deliberative computer (Ethernet). The image processing system consists of a second PC104 embedded computer (PIII 700MHz), a color CCD camera (S-VIDEO, serial interface for control) mounted on a pan/tilt unit (serial), a video transmitter (composite video) and a recorder (miniDV). The deliberative/reactive system runs on a third PC104 embedded computer (PIII 700MHz) which is connected to the PCS with Ethernet using CORBA event channels. The D/R system is described in more detail in the next section.

For further discussion, it is important to note that computational processes are executed concurrently on distributed hardware. Data flow is both synchronous and asynchronous and the concurrent distributed nature of the hardware platform contributes to diverse latencies in data flow throughout the system.

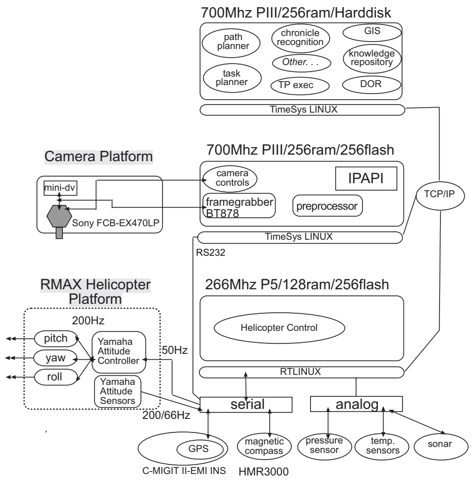

Figure 2. DARA Hardware Schematic

3. DARA: A Distributed Autonomous Robotics Architecture

The DARA system [3] consists of both deliberative and reactive components which interface to the control architecture of the primary flight controller (PFC). Current flight modes include autonomous take-off and landing, pre-defined and dynamic trajectory following and hovering. We have chosen Real-Time CORBA [6][4] as a basis for

2 ”DyKnow” is pronounced as ”Dino” in ”Dinosaur” and stands for *Dynamic Knowledge and Object Structure Processing* .

3 WITAS (pronounced *vee-tas*) is an acronym for the Wallenberg Information Technology and Autonomous Systems Laboratory at Linköping University, Sweden.

4 We are currently using TAO/ACE. **T**he **A**ce **O**rb is an open source implementation of CORBA 2.6.

the design and implementation of a loosely coupled distributed software architecture for our aerial robotic system.

Many of the functionalities which are part of the architecture can be viewed as clients or servers where the communication infrastructure is provided by CORBA facilities and other services such as real-time event channels. Figure 3 depicts an (incomplete) high-level schematic of some of the software components used in the architecture. Each of these may be viewed as a CORBA server/client providing or requesting services from each other and receiving data and events through both real-time and standard event channels.

The modular task architecture (MTA) which is part of DARA is a reactive system design in the procedure-based paradigm developed for loosely coupled heterogeneous systems such as the WITAS aerial robotic system. Reactive behaviors are implemented as *task procedures* (TP) which are essentially event-driven. A TP may open its own (CORBA) event channels, and call its own services (both CORBA and application-oriented services such as path planners) including functionalities from the DyKnow framework. They are executed concurrently.

Given the distributed nature of both the hardware and software architectures in addition to their complexity, one of the main issues is getting data to the right place at the right time in the right form and to be able to transform the data to the proper levels of abstraction for use by high-level deliberative functionalities and middle level reactive functionalities. In the following sections we will describe a very important component of the DARA system which contributes to achieving this.

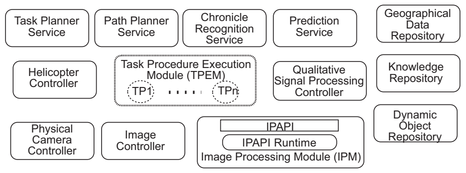
Task Planner Service
Path Planner Service
Chronicle Recognition Service
Prediction Service
Geographical Data Repository
Helicopter Controller
Task Procedure Execution Module (TPEM)
TP1
TPn
Qualitative Signal Processing Controller
Knowledge Repository
Physical Camera Controller
Image Controller
IPAPI
IPAPI Runtime
Image Processing Module (IPM)
Dynamic Object Repository

Figure 3. DARA Software Schematic

4. An Ontological Specification for DyKnow

In any knowledge representation endeavor, it is important to provide a concise specification of the entities or aspects which are the focus of representation. That being said, it is surprisingly difficult to do this in the context of DARA and DyKnow for the following reasons:

- There are often multiple representations of the same entities or aspects associated with those entities. Often this may be known by the system, but equally often it may not.
- The physical locations of representational structures associated with entities must be taken into account due to the distributed and combined asynchronous/synchronous nature of the DARA architecture which often results in unpredictable and varying latencies.
- The dual requirement of not only reasoning about the entities and aspects being represented, but the representational structures themselves since the ultimate purpose of DyKnow is that of situational awareness, getting the right information in the right form to the right place in a timely manner.

Ontologically, we view the external and internal environment of the agent as consisting of entities representing physical and non-physical objects, properties associated with these entities, and relations between entities. We will call such entities *objects* and those properties or relations associated with objects will be called *features*. Features may be static or dynamic and parameterized with objects. Due to the potentially dynamic nature of a feature, that is, its ability to change value through time, a *fluent* is associated with each feature. A fluent is simply a function of time whose range is the feature's type. For a dynamic feature, the fluent values will vary through time, whereas for a static feature the fluent will remain constant through time.

Some examples of features would be the *estimated velocity* of a world object, the *current road segment* of an on-road object, or the *distance between* two car objects. Each fluent associated with these examples implicitly generates a continuous stream of time tagged values of the appropriate type.

Additionally, we will introduce *locations*, *policies*, *computational units* and *fluent streams* which refer to aspects of fluent representations in the actual software architecture. At first glance, this may appear as an ontological level error because software architectures are intended to implement declarative specifications of entities in our ontologies. On the other hand, it is useful to use the representational domain which the architecture implements as a domain of discourse for an autonomous agent if one is interested in reasoning about limited reflective capability as to source, quantity and quality of data flowing through a system and dynamically specifying its use by various functionalities to achieve tasks.

A *location* is intended to denote any pre-defined physical or software location that generates feature data in the DARA architecture. Some examples would be onboard or offboard databases, CORBA event channels, physical sensors or their device interfaces, etc. In fact, a location will be used as an index to reference a representational structure associated with a feature. This structure denotes the process which implements the fluent associated with the feature. A fluent implicitly represents a stream of data, a *fluent stream*. The stream is continuous, but can only ever be approximated in an architecture. A *policy* is intended to represent a particular contextual window or filter used to access a fluent data stream. Particular functionalities in the architecture may need to sample the stream at a particular rate or interpolate values in the stream in a certain manner. Policies will denote such collections of constraints. *Computational units* are intended to denote processes which take fluent streams as input, perform operations on these streams and generate new fluent streams as output.

In summary, the ontology consists of objects, features, fluents, in addition to fluent streams, locations, policies, and computational units. Each of these entities has to be represented either syntactically or in the form of a data structure within the architecture and many of these data structures are grounded through sensor data perceived through the robotic agent's sensors.

An ontologically difficult issue involves the meaning of an object. In a distributed architecture such as DARA, information about a specific object is often distributed throughout the system, some of this information may be redundant and it may often even be inconsistent due to issues of precision and approximation. For example, given a car object, we have seen that it is part of a linkage structure which may contain other *objects* such as onroad, world and vision objects. In addition, many of the features associated with these objects are computed in different manners in different parts of the architecture with different latencies.

One candidate definition for an object could be the aggregate of all feature/value pairs which take the object as a parameter for each feature. It could even be the case that two linkage structures in fact represent the same object in the world but the system is unable to determine this. Two objects may even be of the same type and have linkage structures associated with them, yet the components of the linkage structures may be different. For example, given two car objects, one may not have an onroad object as part of its linkage structure. It is important to point out that objects as intended here have some similarities with OOP objects, but many differences.

5. The DyKnow Framework

In section 4, we considered a number of ontological distinctions that have to be represented within DARA as data structures. In addition, since declarative specifications of both features and policies that determine views of fluent streams are 1st-class citizens in DyKnow, a language for referring to features, locations, computational units and policies has to be provided.

5.1. The Specification Level

Let's begin at the declarative level. We use an extended version of a logic of action and change called TAL [2] to provide the syntax for features and fluents. Any structured primitive feature f is denoted as $\mathsf{f}(loc, policy, arg_1, \ldots, arg_k)$. The parameter loc denotes a system location from a pre-defined set of locations associated with DARA. The location could be a database, an event channel, a sensor, etc. The parameter $policy$ denotes a set of constraints one wishes to place on the feature f's system representation of its associated fluent. The system representation of the fluent will generate a stream of fluent data of the proper type. Often, one may want to control the character of the fluent stream in terms of sampling rate, interpolation strategy, etc. The parameters $arg_1, \ldots, arg_k$ denote the arguments to the feature. For simplicity, we assume these are *object symbols*. These will be described in section 6.1. Here are a few examples:

$\mathsf{estimatedVelocity}(\mathsf{DOR}, \mathsf{policy1}, \mathsf{carObj1})$,
$\mathsf{greenValue}(\mathsf{DOR}, \mathsf{policy3}, \mathsf{visionObj10})$,
$\mathsf{sfs1} = \mathsf{cameraPos}(\mathsf{DOR}, \mathsf{policy4}, \mathsf{cameraObj1})$,
$\mathsf{sfs2} = \mathsf{cameraPos}(\mathsf{camSensor1}, \mathsf{policy5}, \mathsf{cameraObj1})$

Note that in the latter two cases, we are essentially talking about the same property of the onboard camera, in the first via a representation of camera information in the DOR, while in the second via the actual sensor interface.

Additionally, we allow *structured complex features*, whose intended purpose is to denote features defined in terms of others through some operation on the input features. Any structured complex feature f is denoted as $\mathsf{f}(loc, compUnit(fs_1, \ldots, fs_l), policy, arg_1, \ldots, arg_k)$. In this case, the only difference between a structured primitive feature and a structured complex feature is that an additional function representing a computational unit is added, with structured primitive or complex features as arguments. For example, one might like to fuse stream data from the two different sources of data for camera position, denoted sfs1 and sfs2, above. This would be denoted as $\mathsf{cameraPos}(\mathsf{cumod}, \mathsf{compU1}(\mathsf{sfs1}, \mathsf{sfs2}), \mathsf{pol1}, \mathsf{heli})$ where heli refers to the UAV platform and cumod, a computational unit module in DARA.

One could argue that too many aspects of the architecture's implementation level are being lifted into the declarative specification language, but that is just the point. Various high-level functionalities in autonomous systems may often be required to reason about sources, quantity, and quality of data in the absence of operator assistance. Context will also determine what kind of data and how much is required for the task at hand. This particular framework provides the means of reflecting on aspects of the underlying architecture of the system. In fact, in complex distributed environments, autonomous agents may in fact be receiving data streams from outside sources and the location and policy parameters could easily reflect this and be used in the systems reasoning processes.

5.2. The Data Representation Level

The meaning of features, fluents, locations, policies, computational units, and objects is determined through representational structures in DARA. These in turn are grounded through sensors on a continual basis during the achievement of mission tasks. The nature of these representational structures is the topic of this section.

Associated with each primitive structured feature symbol is a *primitive feature representation structure*

$$\langle \mathit{sfs}, \mathit{fluent\ spec.}, \mathit{fluent\ generator}, \mathit{update\ spec.}, \\ \mathit{access\ spec.}, \mathit{cache\ spec.}\rangle$$

A primitive feature representation structure provides information about access and update methods ($\mathit{access\ spec.}, \mathit{update\ spec.}$) for a feature at a location (sfs); storage constraints for samples of the feature ($\mathit{cache\ spec.}$) and constraints ($\mathit{fluent\ spec.}$) on a process ($\mathit{fluent\ generator}$) which can be invoked to generate a stream of data representing the fluent associated with the feature in question. From a semantic perspective, a feature representation structure denotes in some sense, the meaning of the fluent associated with the feature in the sfs. The system can not represent the fluent more precisely due to the inherent limitations of its implementation at this specific location in the architecture.

For example, different types of sensors may place an upper or lower bound on legal sampling capability, or particular types of event channels may only allow push or pull or both push and pull subscription to data. These particular constraints can not be changed by any user of this feature at this location, although if there are choices as to access, these may be further constrained by the representational equivalent of a policy which we will consider shortly. The $\mathit{cache\ specification}$ can specify how many data values are stored or for what interval of time data values should be stored in the cache. For instance, the location may only allow the storage of the current value, or of the current and previous value, or of a bounded stream of values generated in the last 3 seconds.

Associated with each complex structured feature symbol is a *complex feature representation structure*

$$\langle \mathit{sfs}, \mathit{fluent\ spec.}, compUnit(fs_1, \ldots, fs_l), \\ \mathit{access\ spec.}, \mathit{cache\ spec.}\rangle$$

There is little difference between a complex and primitive feature representation structure other than that the fluent generator is replaced with an instantiated computational unit and there is no update specification since the intention is that such structures represent nodes in a data flow network consisting of combinations of streams and computational units. For example, such a structure would be used to translate between coordinates in an image and geographical coordinates.

The internal data structure used to represent a fluent stream is the sample trace, which is a set of samples. Sample traces are generated by the fluent generator or computational unit relative to the sum of constraints in the different specifications. A sample is a triple $\langle t_i, t_s, v\rangle$ consisting of a value and two time-points. The first time-point, the index time, represents the time at which its corresponding value is relevant, e.g. the time a picture was taken from which the position of an object was extracted. The second time-point, the sample time, is the time at which the sample was created. This information is important when reasoning about latencies. The time-points themselves can be sampled either from a discrete or a continuous clock.

Associated with each policy is a *policy descriptor*. A policy descriptor instantiates an interface to a specific feature representation and can be used to place further constraints on the fluent stream generated by the specific feature representation. Any functionality in DARA that wants to access a specific feature must do so through a policy descriptor. The policy descriptor is thus contextual in nature and there can be many users of a specific feature representation, each with its own unique policy. There are many uses of policy descriptors. For instance, they can be used to implement specific sampling rates and when specific values aren't provided by the associated feature representation, the policy descriptor can interpolate stream values in different ways to assure values for each sample. A descriptor can be used as a filter and only generate values within upper and/or lower bounds, or only generate values when there is a change in the stream. In the forthcoming examples, some of these techniques will be demonstrated.

5.3. The Semantic Level

The specification level provides the DARA system with a logical tool consisting of an extended version of a logic of action and change, TAL (temporal action logic) which can be used to specify features used in DARA, data flow networks constructed from computational units and fluent streams, and policies which contextually modify fluent streams for particular constraints associated with mission tasks. In addition, TAL can be used as a basis for plan generation, execution monitoring, diagnosis, and chronicle recognition, although currently, it is primarily used for specification purposes.

The data representation level implements the DyKnow framework which relates the formal specification to sensory input via a suite of representational structures described in the previous section. The data representation level can also be used separately from the specification level by the different deliberative, reactive and control functionalities in the DARA system.

In fact, one can view DyKnow as implementing a distributed qualitative signal processing tool where the system is given the functionality to generate dynamic representations of parts of its internal and external environment in a contextual manner through the use of policy descriptors and feature representation structures. The dynamic representations can be viewed as collections of time series data at various levels of abstraction, each time series representing a particular feature and each bundle representing a particular history or progression. Another view of such dynamic representations and one which is actually put to good use is to interpret the fluent stream bundles as partial temporal models in the logical sense. These partial temporal models can then be used on the fly to interpret temporal logical formulas in TAL or other temporal formalisms. Such a functionality can be put to good use in constructing execution monitors, predictive modules, diagnostic modules, etc. The net result is a very power mechanism for dealing with a plethora of issues associated with focus of attention and situational awareness.

6. Applications using DyKnow

In the following two subsections, we will show how the DyKnow framework can be used to generate fluent streams for further processing by two important deliberative functionalities in the DARA system, chronicle recognition and execution monitoring. Both are implemented in the UAV system. Before doing this, we provide a short description of the Dynamic Object Repository (DOR), an essential part of the DARA which uses the DyKnow framework to provide other functionalities in the system with information about the properties of dynamic objects most often constructed from sensor data streams.

6.1. The Dynamic Object Repository

The Dynamic Object Repository (DOR) is essentially a soft realtime database used to construct and manage the object linkage structures described in the introduction. The DOR is implemented as a CORBA server and the image processing module interfaces to the DOR and supplies vision objects. Task procedures in the MTA access feature information about these objects via the DyKnow framework, creating descriptors on-the-fly and constructing linkages. Computational units are used to provide values for more abstract feature properties associated with these objects. For example, the co-location process involving features from the vision, helicopter and camera objects, in addition to information from the GIS, use computational units to output geographical coordinates. These are then used to update the positional features in world objects linked to the specific vision objects in question.

Objects are referenced via unique symbols which are created by the symbol generation module which is part of the DOR. Each symbol is typed using pre-defined domains such as car, world-object, vision-object, vehicle, etc. Symbols can be members of more than one domain and are used to instantiate feature representations and as indexes for collecting information about features which take these symbols as arguments. Since domains collect symbols which *reference* a certain type of object, one can also conveniently ask for information about collections or aggregates of objects. For example, “take all vision objects and process a particular feature for each in a certain manner”.

6.2. An Application to Chronicle Recognition

Chronicles are used to represent complex occurrences of activity described in terms of temporally constrained event structures. In this context, an event is defined as a change in value of a feature. For example, in a traffic monitoring application, a UAV might fly to an intersection and try and identify how many vehicles turn left, right or drive straight through a specific intersection. In another scenario, the UAV may be interested in identifying vehicle overtaking. Each of these complex activities can be defined in terms of one or more chronicles. In the WITAS UAV, we use the CRS. chronicle recognition system developed by France Telecom. CRS is an extension of IxTeT [4]. Our chronicle recognition module is wrapped as a CORBA server.

As an example, suppose we would like to recognize vehicles passing through an intersection. Assume cars are being identified and tracked through the UAV’s camera as it hovers over a particular intersection. Recall that the DOR generates and maintains linkage structures for vehicles as they are identified and tracked. It can be assumed that the following structured features exist:

pos = position(DOR, policy1, car1)
roadseg = road_segment(DOR, roadSegment(pos), policy2, car1)
incross = in_crossing(DOR, inCrossing(roadseg), policy3, car1)

pos is a feature of a car object and its fluent stream can be accessed via the DOR as part of its linkage structure. roadseg is a complex feature whose value is calculated via a computational unit roadSegment which takes the geographical position of a world object associated with the car object as argument and uses this as an index into the GIS to return the road segment that the vehicle is in. Similarly, incross is a complex feature structure whose boolean value is calculated by using a computational unit that takes the roadSegment fluent stream as input and returns a boolean output stream calculated via a lookup in the GIS.

For the sake of brevity, a car is defined to pass through an intersection if its road segment type is not a crossing then it eventually is in a road segment that is a crossing and then it is again in a road segment that is not a crossing. In this case, if the fluent stream generated by incross generates samples going from false to true and then eventually true to false within a certain time frame then the car is recognized as passing through a crossing. The chronicle recognition system would receive such streams and recognize two change events which match its chronicle definition.

The stream itself requires some modification and policy3 specifies this via a *monotonic time* constraint and a *change* constraint. The monotonic time constraint would make sure the stream is ordered, i.e. the time stamp of events increase monotonicly . The change constraint specifies how change is defined for this stream. There are several alternatives which can be used:

- *any change policy* – any difference between the previous and current value is a change;
- *absolute change policy* – an absolute difference between the previous and current value larger than a parameter $delta$ is a change;
- *relative change policy* – a normalized difference between the previous and current value larger than a parameter $delta$ is a change.

There are obvious variations on these policies which are intended to deal with oscillatory values due to uncertainty of data, etc., and definitions of chronicles can become quite complex. This example is only intended to provide an overview as to how DyKnow is used by other modules.

6.3. An Application to Execution Monitoring

The WITAS UAV architecture has an execution monitoring module which is based on the use of a temporal logic, LTL (linear temporal logic with intervals [5]), which provides a succinct syntax for expressing highly complex temporal constraints on activity in the UAV's internal environment and even aspects of its embedding environment. For example safety and liveness conditions can easily be expressed. Due to page limitations we can only briefly describe this functionality. Essentially, we appeal to the intuitions about viewing bundles of fluent streams as partial models for a temporal logic and evaluating formulas relative to this model. In this case though, the model is fed piecewise (state-wise) to the execution monitor via a state extraction mechanism associated with the execution monitor. A special progression algorithm [5] is used which evaluates formulas in a current state and returns a new formula which if true on the future states would imply that the formula is true for the complete time-line being generated.

The DyKnow system is ideal for generating such streams and feeds these to the execution monitor. Suppose we would like to make sure that two task procedures (all invocations) in the reactive layer of the DARA, called A and B, can never execute in parallel. For example, A and B may both want to use the camera resource. This safety condition can be expressed in LTL as the temporal formula G¬(∃x∃y tp_name[x]=A ∧ tp_running[x]=true ∧ tp_name[y]=B ∧ tp_runing[y]=true), where "G" in the formula is the modal operator for "at all times". To monitor this condition the execution monitor requires fluent streams for each of the possible instantiations of the parameterized features tp_name and tp_running which can be generated by the reactive layer of the DARA. These are fed to the instantiated execution monitor which applies the progression algorithm to the temporal formula above relative to the fluent streams generated via the DyKnow framework. This algorithm is run continuously. If the formula evaluates to false at some point, an alert message is sent to a monitor set up by the functionality interested in this information and modifications in the system configuration can be made.

References

[1] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 747–755, 2000.

[2] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. (TAL) temporal action logics: Language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2(3-4):273–306, 1998. http://www.ep.liu.se/ej/etai/1998/009/.

[3] P. Doherty, P. Haslum, T. Merz, E.Skarman, G. Conte, S. Duranti, F. Heintz, P.-O. Pettersson, T. Persson, and B. Wingman. A distributed architecture for intelligent unmanned aerial vehicle experimentation. 2004. Submitted to ECAI 04.

[4] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR-96)*, 1996.

[5] K. B. Lamine and F. Kabanza. Reasoning about robot actions: A model checking approach. In *Advances in Plan-Based Control of Robotic Agents*, LNAI, pages 123–139, 2002.

[6] Object Computing, Inc. *TAO Developer's Guide, Version 1.1a*, 2000. See also `http://www.cs.wustl.edu/~schmidt/TAO.html`.