Managing Dynamic Object Structures using Hypothesis Generation and Validation

Fredrik Heintz and **Patrick Doherty**

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. In this paper, we focus on the use of DyKnow in supporting the representation and reasoning about dynamic objects such as road vehicles in the external environment of an autonomous unmanned aerial vehicle. The representation of complex objects generally consists of simpler objects with associated features that are related to each other via linkages. These linkage structures are constructed incrementally as additional sensor data is acquired and integrated with existing structures. The resulting linkage structures represent complex objects at many levels of abstraction. Many issues related to anchoring and symbol grounding can be approached by taking advantage of the versatility of these linkage structures. Examples are provided in the paper using an experimental UAV research platform.

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 robotic agent is often both static and dynamic. A great amount of background or deep knowledge is required by the agent in understanding its world and in understanding 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 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. How to define a change, how to approximate values at time-points where no value is given and how to synchronize collections of values are examples of properties that can be set in the context. 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, and execution monitoring.

The paper is structured as follows. We start with a section where a larger scenario using the proposed techniques is described. The next section provides a description of the UAV platform used in our experiments. The third section describes a distributed autonomous robotics architecture developed to support the integration of deliberative, reactive and control functionalities. In the fourth section the DyKnow framework itself is introduced. The fifth section concludes with a description of hypothesis generation and validation mechanisms which are used to create and manage dynamic object structures.

An Identification and Track Scenario

In order to make the 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.

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.

One way for the UAV to achieve its task would be to initiate a reactive task procedure (parent procedure) which calls the image processing module with the vehicle signature as a parameter. The image processing module will try to identify colored blobs in the region of the right size, shape and color as a first step. The features of each new blob, such as RGB values with uncertainty bounds, length and width in pixels and position in the image, are associated with a *vision object*. The image processing system will then try to track the blobs. 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.

Now one can hypothesize that the blob actually exists in the world and represent a single entity by creating a representation of the blob in the world. New features, such as position in geographical coordinates, are associated with a new *world object*. The geographic coordinates provide a common frame of reference where positions over time and over different objects can be compared. To represent that the two objects represent two aspects of the same entity the vision object is *linked* to the world object. Since the two objects are related the features of the world object will be computed from features of the linked vision object. 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.

Assuming the UAV only has one camera, the link from a vision object to a world object will be one-to-many, i.e. several world objects could be hypothesized from the same vision object but each world object only depends on one vision object. If there was more than one camera then more than one vision object could be associated with the world object, one for each camera. To compute the features of the world object, computations and fusion between feature values from the linked-from objects would be required.

Each time a new vision object is created, it is tested against each existing world object to see if they could represent the same entity. If the world object passes the test then a link is created between it and the vision object. In this case, world object features would be updated using features from the new vision object as long as they remain linked. This is an example where the world object has been reacquired.

Since links only represent hypotheses, they are always subject to becoming invalid given additional data, so the UAV agent continually has to verify the links validity. This is done by associating maintenance constraints with links which must continually be monitored for validity. A maintenance constraint could compare the behavior of the new entity, which is the combination of the two representations, with the normative behavior of this type of entity and, if available, the predicted behavior of the previous entity.

The next qualitative step in creating a linkage structure in this scenario would be to check if the world object is a moving object. In this case, it would be hypothesized if the world object's position feature changes over time. The same condition can be used to maintain the link. Failure of this condition would depend on further hypotheses and there are many choices. For example, if there is a further hypothesis where the object is considered to be a vehicle, when it stops, it may be just to park, and it should retain its moving vehicle status.

To continue the chain of qualitative levels of representations, if the moving object is on or close to a road, as defined by a geographical information system (GIS), then we could hypothesize that it's an *on-road object*, i.e. an object moving along roads. The maintenance condition is that it's actually following the road system, otherwise it would be an *off-road object* (which we ignore in this scenario). 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 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 last step in our qualitative representation of entities is to hypothesize what kind of vehicle it is. The default assumption is that it's a car, but if it's too large or too small then one could hypothesize that it's a truck, bus or motorcycle. Here it is assumed that background knowledge about vehicle types exists and can be put to use in determining vehicle type.

All object types, links and constraints are currently configured by a parent task procedure at the beginning of an identification scenario. Thus if the situation changes the task procedure has the option of modifying the object and link specifications associated with the task at hand.

How then can a robotic's architecture, in particular, the UAV architecture described here, be set up to support the processes described in the UAV scenario above? In (Heintz & Doherty 2004) a software system called the *DyKnow Framework*[2] is proposed for supporting the use of dynamic

[1] GPS and INS are acronyms for global positioning system and inertial navigation system, respectively.

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

knowledge structures. In this paper we extend the framework with techniques for the management of dynamic object structures as described above.

The WITAS UAV Platform

The WITAS[3] Unmanned Aerial Vehicle Project (Doherty *et al.* 2000; Doherty 2004) is a 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 26 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 other PCs with Ethernet using CORBA event channels. The D/R system is described in more detail in the next section.

DARA: A Distributed Autonomous Robotics Architecture

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

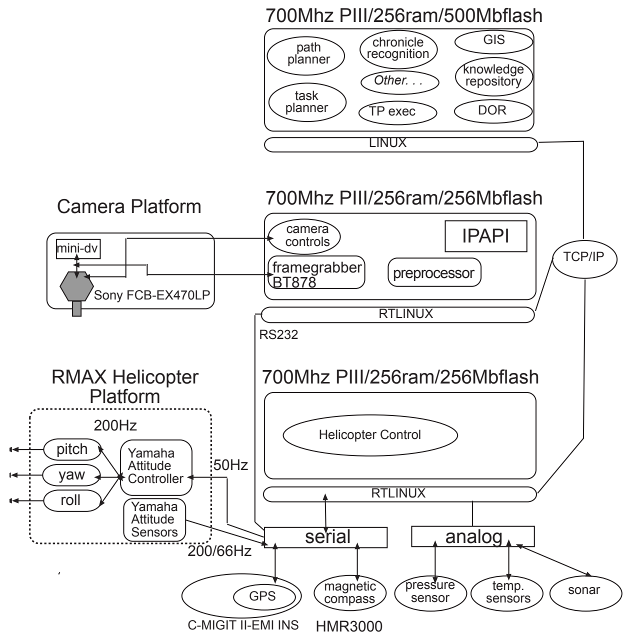

Figure 2: DARA Hardware Schematic

The DARA system (Doherty *et al.* 2004) 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, vehicle tracking and hovering. We have chosen Real-Time CORBA (Object Computing, Inc. 2003)[4] as a basis for 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 executed concurrently and 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 in DyKnow.

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

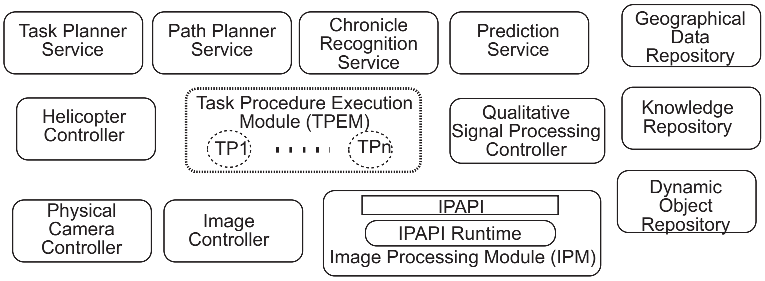

Figure 3: DARA Software Schematic

DyKnow

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. DyKnow is designed to contribute to achieving this.

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 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, and 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. 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. 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. Each of these entities are 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. 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 is provided, see (Heintz & Doherty 2004) for details.

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 (temporal action logic) 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 powerful mechanism for dealing with a plethora of issues associated with focus of attention and situational awareness.

Dynamic Object Structure in DyKnow

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, it can be part of a linkage structure which may contain other *objects* such as on-road, world and vision objects. For an example of a linkage structure see figure 4. 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 features which take the object as a parameter for each feature. But an object only represents some aspects of an entity in the world. To represent that several different objects actually represent the same entity in the world, links are created between those objects. It is these linkage structures that represent all the aspects of an entity which are known to the UAV agent. It can be the case that two linkage structures in fact represent the same entity in the world but the UAV agent is unable to determine this. Two objects may even be of the same type but have different linkage structures associated with them. For example, given two car objects, one may not have an on-road object, but an off-road 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.

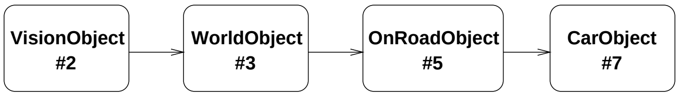

Figure 4: An example object linkage structure

Hypothesis Generation

Each object is associated with a set of possible hypotheses, each associated with constraints relating the object to another object. Each possible hypothesis is a relation between two objects associated with constraints between the objects. To generate a hypothesis, the constraints of a possible hypothesis must be satisfied. Two different types of hypotheses can be made depending on the types of the objects. If the objects have different types then a hypothesis between them is represented by a link. If they have the same type then a hypothesis is represented by a codesignation between the objects. Codesignations hypothesize that two objects representing the same aspect of the world are actually identical, while a link hypothesizes that two objects represent different aspects of the same entity.

A link can be hypothesized when a reestablish constraint between two existing objects is satisfied or an establish constraint between an object and a newly created object is satisfied. In the anchoring literature these two processes are called reacquire and find (Coradeschi & Saffiotti 2003).

Since the UAV agent can never be sure its hypotheses are true, it has to continually verify and validate them against its current knowledge of the world. To do this, each hypothesis is associated with maintenance constraints which should be satisfied as long as the hypothesis holds. If the constraints are violated then the hypothesis is removed. The maintenance and hypothesis generation constraints are represented using the linear temporal logic (LTL) with intervals (Lamine & Kabanza 2002) and are checked using an execution monitoring module which is part of the DyKnow framework (Heintz & Doherty 2004).

Object Specification

Before DyKnow can generate object structures, the controlling task procedure has to specify the appropriate links and object types. The latter are called classes. A link specification has two parts, the classes which the link associates and the constraint specifications for establishing and maintaining instances of the link. A link l is denoted as $\mathsf{l}(from, to, establish, reestablish, maintain)$, where $from$ and to are the names of classes whose instances the link associates, the three constraints $establish$, $reestablish$, and $maintain$ specify when to create and delete instances of the link. The constraints are expressed as LTL formulas containing features associated with the objects being linked. In order to refer to the objects being linked, the special feature arguments to and $from$ can be used.

A class specification consists of the specifications of the links and features associated with the class, the delete constraint and the codesignation policy. A class C is denoted as $\mathsf{C}(\{l_1, \ldots, l_n\}, \{f_1, \ldots, f_m\}, delete, codesignation)$, where l_i is a link specification and f_j is a feature specification with special arguments representing the actual object instance ($this$) and each incoming link (the name of the link). The delete constraint specifies when an instance of the class should be deleted. The codesignation policy consists of the codesignation strategy, the codesignation constraint and the merge function. It specifies when two objects belonging to the class should be hypothesized as being identical and how to merge them. The constraints are expressed as LTL (with intervals) formulas using feature specifications with the special argument $this$, which represents the instance of the class. Two example classes are:

```
VisionObject({},
 {mx=mean_x(DOR, pol1, this),
  my=mean_y(DOR, pol1, this), ...},
 not( always(mx == prev(mx)
             until[0..30]
             mx != prev(mx) ) ),
 <keep_old, false, f>)

WorldObject({vo_wo_link(
 VisionObject, WorldObject,
 eventually(vo_wo_link(DOM, pol2, from)
               = {}),
 eventually(dist_est_pos(DOR,
               pol3, from, to) < 5),
 true)},
 {p=position(DOR,
     colocate(mean_x(DOR, pol4, vo_wo_link),
              mean_y(DOR, pol4, vo_wo_link)),
     pol5, this), ...},
 not( always(p == prev(p)
             until[0..60]
             p != prev(p))),
 <keep_old, false, g>)
```

The VisionObject class specification states that a VisionObject has two features with the specifications $\mathsf{mean\_x}(\mathsf{DOR}, \mathsf{pol1}, this)$ and $\mathsf{mean\_y}(\mathsf{DOR}, \mathsf{pol1}, this)$, where DOR is the feature location which should host the feature representations, this will be replaced with the actual object identifier. $\mathsf{pol1}$ is the policy used to create the associated fluents for the feature representations. The delete constraint states that the object should be deleted when the $\mathsf{mean\_x}$ feature hasn't been updated within 30 seconds. The WorldObject class specification says that a WorldObject has one feature with the specification $\mathsf{position}(\mathsf{DOR}, \mathsf{colocate}(\mathsf{mean\_x}(\mathsf{DOR}, \mathsf{pol3}, vo\_wo\_link), \mathsf{mean\_y}(\mathsf{DOR}, \mathsf{pol3}, vo\_wo\_link)), \mathsf{pol4}, this)$, where vo_wo_link will be replaced with the object identifier for the linked from VisionObject instance. If the link is not established no position feature will be created. The specification states that the position is calculated from the vision coordinates taken from the linked from VisionObject.

Object Management

Objects are created either explicitly or as part of hypothesis generation. In either case the object manager is responsible for creating the object. To create an object instance of a class it has to instantiate the links and features and create monitors for the establish, reestablish, codesignation and delete constraints associated with the new object. When the object instance is deleted, either explicitly or because the delete constraint is satisfied, the feature representations have to be deleted together with all monitors and links associated with the object. The object manager is also responsible for keeping track of the links objects are connected to and instantiating features when new links have been created since the features of an object may be dependent on features related to linked-from objects.

Algorithm for creating an object instance of a class C

```
1. create instances of all features f
   related to the class C
2. create a monitor for the delete
   constraint for the class C
3. for each other object instance o of the
   class C create a monitor for the
   codesignation constraint for C between
   this and o
4. for each link L which links from the
   class C do
  a. create a new object instance o of
     the linked to class D and create
     a monitor for the establish
     constraint for L between this and o
  b. for each object instance o of the
     linked to class D create a copy o'
     and a monitor for the reestablish
     constraint for L between this and o'
5. for each link L which links to the
   class C do
  a. for each object instance o of the
     linked from class B create a monitor
     for the reestablish constraint for
     L between o and this
```

Link Management

When establish or reestablish constraints are satisfied the object manager should create a link between two objects. If there is no link of the same type to the linked-to object then the new link is added and returned to the object manager, otherwise the existing link and the new link have to be merged since an object is only allowed to be linked to once for each link type. Three strategies are used to merge two links:

1. apply a select function which compares the two links and returns the link representing the best hypothesis; or
2. assume the linked from objects represent different versions of the same aspect of the same entity, since they are not codesignated, and merge the two links to a many-to-one link; or
3. duplicate the linked to object and link to the new object, i.e. we would have two different hypotheses about the same object (i.e. a disjunctive hypothesis).

Figure 5 shows the result after linking VisionObject #11 to WorldObject #3 (which is already linked to VisionObject #2) using the second (top) and third strategy (bottom).

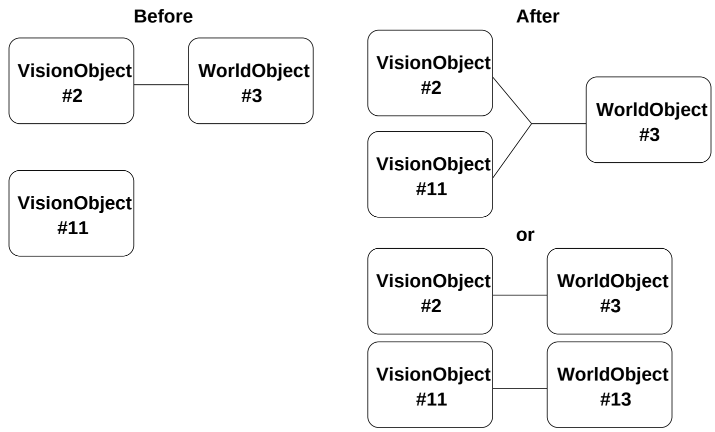

Figure 5: Two examples of merging links

The many-to-one link used above can be realized in different ways depending on the properties of the objects. The purpose is to merge the information stored in n objects from class A to a single object of class B. To do this we can either create a single object of class A representing all the n objects or we can create n instances of the class B and then merge these n instances to a single object from class B. The merging of the objects is done using the merge function in the codesignation policy of the class. This implies that either A or B has to have a merge function defined otherwise the many-to-one link can't be realized. The second option, i.e. to create n instances of B and then merge these, also requires that it is possible to create an instance of B from an instance of A. The default approach is to merge the n instances of A to a single instance and use it in the link.

Algorithm for establishing a link between a and b:

```
1. Create a link instance between a and b
2. Merge existing links
3. Instantiate the features related to the
   link in a and b
4. Set up a monitor for the maintain
   constraint on the link
```

Codesignation Management

When objects are hypothesized as being identical the object manager has to merge them into one representation. This can be done in two ways, either merge the objects into one of the existing objects (and delete the others) or create a new object which is the result of the merge. Which approach to use is defined by the strategy in the codesignation policy. In either case we need to merge the information stored in the objects and this is done with the merge function which is also a component in the codesignation policy. Finally the links related to the objects have to be merged. All outgoing links have to be moved to the merged object and duplicated links removed. If we keep the old objects and continually update a new merged object from them we also keep the incoming links as they are, otherwise we have to move them to the object we are keeping and merge them as described above. A benefit of the strategy to create a new object which is the result of continually merging two or more objects is that the codesignation hypothesis can be withdrawn if it's no longer valid. The drawback is less performance and duplication of data.

Lifecycle Example

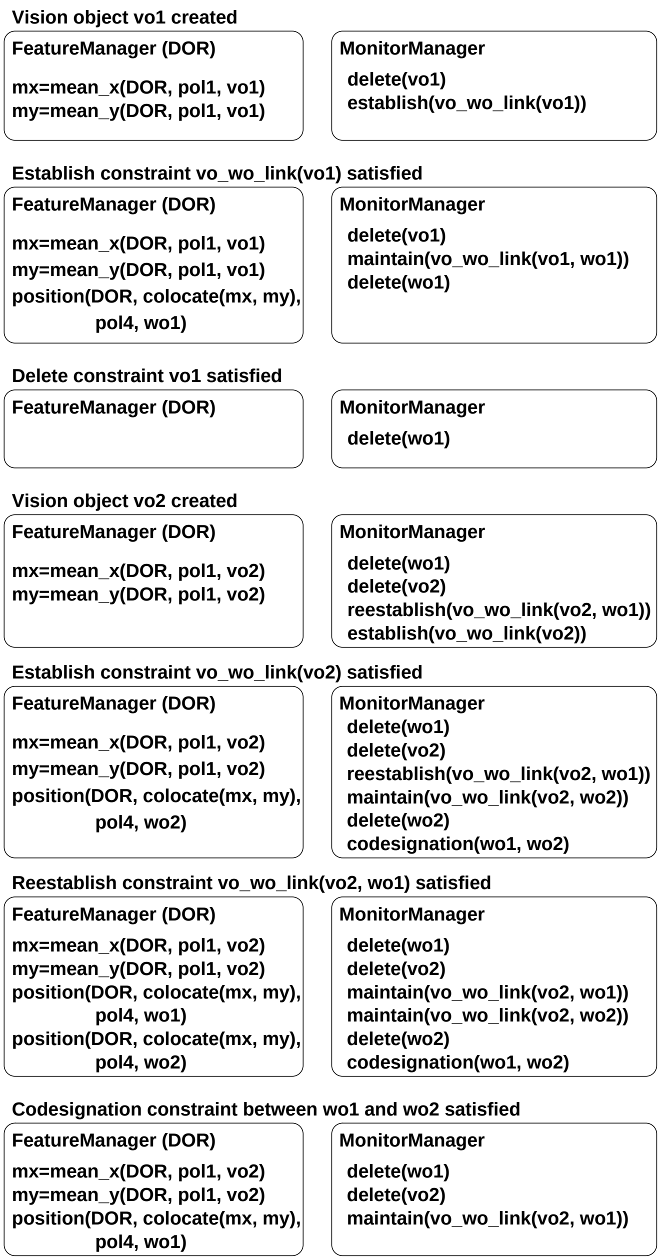

Figure 6: The state of the feature and monitor managers during the example.

Using the VisionObject and WorldObject specifications above, assume a VisionObject vo1 is created. The mean_x(DOR, pol1, vo1) and mean_y(DOR, pol1, vo1) feature representations are created, as well as a delete constraint monitor for vo1. Since a VisionObject can participate in a `vo_wo_link` link an establish monitor is created. Assume the establish constraint is satisfied, then a WorldObject, e.g. wo1, is created. Together with it the feature representations mx = mean_x(DOR, pol3, vo1), my = mean_y(DOR, pol3, vo1), and position(DOR, colocate(mx, my), pol4, wo1) are created. A delete monitor for the object is created as well as a maintenance monitor for the link between vo1 and wo1. The states of the feature and monitor managers at the different stages of the example are shown in figure 6.

Now assume the tracking of vo1 is lost and the object removed. This means the link between vo1 and wo1 is also removed, but not wo1 itself, even though it will no longer be updated. A few images later a new blob is found and a new VisionObject vo2 is created. Then the mean_x and mean_y feature representations are created, together with monitors for the delete, `vo_wo_link` establish and `vo_wo_link`(vo2, wo1) reestablish constraints. If the establish constraint is satisfied then a new WorldObject, e.g. wo2, is created together with the feature representations for position as well as monitors for the delete and codesignate(wo1, wo2) constraints. After some time the reestablish constraint might be satisfied and a link between vo2 and wo1 is created together with a maintenance monitor. Later the codesignate constraint between wo1 and wo2 is satisfied, which is natural since they are computed from the same vision object, wo1 and wo2 are merged into wo1 and wo2 is removed.

Related Work

In this section, we will compare three approaches to anchoring symbols which use techniques having some similarity to the DyKnow approach. According to (Coradeschi & Saffiotti 2003), anchoring is "the process of creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects".

Instead of creating one symbol for each physical object as described in (Coradeschi & Saffiotti 2000; 2001) we create one symbol for each aspect of a physical object we are interested in, working in a bottom-up fashion from sensor data. These symbols can then be linked together to assert the fact that they actually represent the same entity in the world. The benefit is that we do not have to create one function for determining whether the blob seen in an image is a car but rather can split up the problem into smaller and simpler problems, like determining whether a blob is an object in the world, if a world object is moving along roads and finally whether an object moving along roads is a car.

Another benefit is that we can generate several hypothesized objects at different levels of abstraction, which are all anchored to the same sensor data, to handle uncertainty in object identities. To verify and validate the hypotheses constraints are placed on the links in the linkage structures and monitors are created to continually check that constraints are not violated. If the constraints on a link are violated, the link is removed and the particular object is no longer grounded. The object itself is not removed since we might reestablish a link at a later time. This corresponds to a reacquire in the terminology of (Coradeschi & Saffiotti 2003). There are two ways of relating our linkage structures to their anchors, either view each linkage structure as an anchor or each link as an anchor and the linkage structure as a chain of anchors. Either way we have great flexibility in describing the requirements for creating and maintaining the anchors.

Another related approach is (Fritsch *et al.* 2003) where they propose a method for anchoring symbols denoting composite objects through anchoring the symbols of their corresponding component objects. They extend the framework presented by Coradeschi and Saffiotti with the concept of a composite anchor which is an anchor without a direct perceptual counterpart. Instead the composite anchor computes its own perceptual signature from the perceptual signatures of its component objects. The benefit is that each sensor can anchor its percepts to symbols which can be used to build composite objects fusing information from several sensors. The same functionality can be provided by DyKnow since objects do not have to have direct perceptual counterparts, but can be computed from other objects which may or may not acquire their input directly from sensors.

This particular functionality is important to emphasize since in complex hybrid robotic architectures, different components and functionalities in the architecture require access to representations of dynamic objects in the external environment at different levels of abstraction and with different guaranteed upper bounds on latencies in data. By modeling dynamic objects as structured objects with different types of features, any functionality in the architecture can access an object at the proper level of abstraction and acquire data from the object in a timely manner.

A final related approach is that of (Bonarini, Matteucci, & Restelli 2001), where they use concepts with properties to model objects. They introduce a model which is the set of all concepts linked by relationships. The relationships can represent constraints that must be satisfied, functions which generate property values for a concept from property values of another concept, or structural constraints which can be used to guide anchoring (such as the fact that two concepts are a total and exclusive specialization of another concept).

In DyKnow such functions are called computational units and the constraints used are partitioned into several types depending on their function. Although we do not have direct support for structural constraints, we can use existing DyKnow functionality to represent disjuncts such as the fact that a moving object is either an off-road object or an on-road object but not both.

Another difference between the approaches is that Bonarini et al compute the degree of matching for each concept in order to better handle uncertain and incomplete information. Similarity measurements between objects are an essential functionality for anchoring objects to sensor data and comparing them to each other. Currently, we have developed a general theory for measuring similarity based on the use of rough set techniques (Doherty & Szałas 2004; Doherty, Łukaszewicz, & Szałas 2003; Doherty & Szałas 2004), although this particular functionality has not yet been integrated into DyKnow. This is part of our ongoing activity in this area.

Summary

DyKnow is a software framework developed for supporting access to signal data in robotic architectures at various levels of abstraction and fusing such data into component objects representing entities in both the internal and external environments of robotic systems. In this particular case we focus on an unmanned aerial vehicle and describe functionality which supports contextual access, creation, storing and processing dynamic objects representing vehicles perceived by the UAV. This subset of DyKnow functionalities provides a generic toolkit for dealing with many issues related to anchoring and symbol grounding in robotic systems. We describe this subset and compare this functionality to related work. The UAV and DyKnow system have been tested experimentally in actual flights.

Acknowledgements

This work is partially supported by a grant from the Wallenberg Foundation, Sweden and NFFP 539 COMPAS.

References

Bonarini, A.; Matteucci, M.; and Restelli, M. 2001. Anchoring: do we need new solutions to an old problem or do we have old solutions for a new problem? In *Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems: Papers from the 2001 AAAI Fall Symposium, Technical Report No. FS-01-01*, 79–86. Menlo Park, CA: AAAI Press.

Coradeschi, S., and Saffiotti, A. 2000. Anchoring symbols to sensor data: preliminary report. In *Proc. of the 17th AAAI Conf.*, 129–135. Menlo Park, CA: AAAI Press.

Coradeschi, S., and Saffiotti, A. 2001. Perceptual anchoring of symbols for action. In *Proc. of the 17th IJCAI Conf.*, 407–412.

Coradeschi, S., and Saffiotti, A. 2003. An introduction to the anchoring problem. *Robotics and Autonomous Systems* 43(2-3):85–96.

Doherty, P., and Szałas, A. 2004. On the correspondence between approximations and similarity. In *Proceedings of the 4th International Conference on Rough Sets and Current Trends in Computing, RSCTC'2004*.

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

Doherty, P.; Haslum, P.; Heintz, F.; Merz, T.; Nyblom, P.; Persson, T.; and Wingman, B. 2004. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*.

Doherty, P.; Łukaszewicz, W.; and Szałas, A. 2003. Tolerance spaces and approximative representational structures. In *Proceedings of the 26th German Conference on Artificial Intelligence*.

Doherty, P. 2004. Advanced research with autonomous unmanned aerial vehicles. In *Proceedings on the 9th International Conference on Principles of Knowledge Representation and Reasoning*.

Fritsch, J.; Kleinehagenbrock, M.; Lang, S.; Pltz, T.; Fink, G. A.; and Sagerer, G. 2003. Multi-modal anchoring for human-robot interaction. *Robotics and Autonomous Systems* 43(2-3):133–147.

Heintz, F., and Doherty, P. 2004. DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems*.

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

Object Computing, Inc. 2003. *TAO Developer's Guide, Version 1.3a*. See also `http://www.cs.wustl.edu/~schmidt/TAO.html`.