Stream-Based Middleware Support for Embedded Reasoning*

Fredrik Heintz, Jonas Kvarnström and **Patrick Doherty**
Dept. of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden
{frehe, jonkv, patdo}@ida.liu.se

Abstract

For autonomous systems such as unmanned aerial vehicles to successfully perform complex missions, a great deal of embedded reasoning is required at varying levels of abstraction. In order to make use of diverse reasoning modules in such systems, issues of integration such as sensor data flow and information flow between such modules has to be taken into account. The DyKnow framework is a tool with a formal basis that pragmatically deals with many of the architectural issues which arise in such systems. This includes a systematic stream-based method for handling the sense-reasoning gap, caused by the wide difference in abstraction levels between the noisy data generally available from sensors and the symbolic, semantically meaningful information required by many high-level reasoning modules. DyKnow has proven to be quite robust and widely applicable to different aspects of hybrid software architectures for robotics.

In this paper, we describe the DyKnow framework and show how it is integrated and used in unmanned aerial vehicle systems developed in our group. In particular, we focus on issues pertaining to the sense-reasoning gap and the symbol grounding problem and the use of DyKnow as a means of generating semantic structures representing situational awareness for such systems. We also discuss the use of DyKnow in the context of automated planning, in particular execution monitoring.

Introduction

For autonomous systems such as unmanned aerial vehicles (UAVs) to successfully perform complex missions, a great deal of embedded reasoning is required. In order for this reasoning to be grounded in the environment, it must be firmly based on information gathered through the 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. We call this the *sense-reasoning gap*.

Bridging this gap is a prerequisite for essential 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.

However, creating a suitable bridge is a very challenging problem. It requires constructing suitable 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 as well as application-specific approaches.

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. It also becomes less practical to statically predefine exactly how the information processing should be configured. Instead it is necessary to dynamically tailor the processing based on the current task and requirements. When the task or the requirements change the configuration of the processing should change accordingly. It also becomes harder to predict the computational load. Since always considering the worst case would waste too much resources, the system must adapt the processing at run-time to manage the fluctuating load.

To allow the sense-reasoning gap to be efficiently bridged on a distributed robotic platform these issues must be taken into consideration. An important problem is therefore how to configure the manner in which information and knowledge is processed and reasoned about in a context-dependent manner to achieve high-level goals while globally optimizing the use of resources and the quality of the results.

To address these issues we have developed the stream-based knowledge processing middleware framework Dy-

*This work is partially supported by grants from the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR) Linnaeus Center CADICS, and the Center for Industrial Information Technology CENIIT (projects 06.09 and 10.04).

Know (Heintz & Doherty 2004; Heintz 2009) which is a central component of our UAV architecture (Doherty *et al.* 2004; Doherty 2004).

Desired Properties

A wide range of functionality could conceivably be provided by middleware for embedded reasoning, and no single definition will be suitable for all systems. As a starting point, we present the requirements that have guided our work in distributed UAV architectures. These requirements are not binary in the sense that a system either satisfies them or not. Instead, a system will satisfy each requirement to some degree. Later, we will argue that DyKnow provides a significant degree of support for each of the requirements.

Support integration of existing reasoning functionality. The most fundamental property of middleware is that it supports interoperability. In the case of knowledge processing middleware, the main goal is to facilitate the integration of a wide variety of existing reasoning engines and sensors, bridging the gap between the distinct types of information required by and produced by such entities.

Support distributed sources and processing. Knowledge processing middleware should permit the integration of information from distributed sources and the distribution of processing across multiple computers. For UAVs, sources may include color and thermal cameras, GPS sensors, and laser range scanners as well as higher level geographical information systems and declarative specifications of objects and their normal behaviors. Knowledge processing middleware should be sufficiently flexible to allow the integration of such sources into a coherent processing system while minimizing restrictions on connection topologies and the type of information being processed.

Support quantitative and qualitative processing on many levels of abstraction. In many applications there is a natural information abstraction hierarchy starting with quantitative signals from sensors, through image processing and anchoring, to representations of objects with both qualitative and quantitative attributes, to high level events and situations where objects have complex spatial and temporal relations. Some of this information is quantitative, while some is qualitative. It should be possible to process information having arbitrary forms at arbitrary levels of abstraction, incrementally transforming it to forms suitable for various types of low-level and high-level reasoning.

Support bottom-up data processing and top-down model-based processing. While each process can be dependent on "lower level" processes for its input, it should also be possible for its output to guide processing in a top-down fashion. For example, if a vehicle is detected on a particular road segment, a vehicle model could be used to predict possible future locations, thereby directing or constraining processing on lower levels.

Support management of uncertainty. Uncertainty exists not only at the quantitative sensor data level but also in the symbolic identity of objects and in temporal and spatial aspects of events and situations. Therefore, middleware should not be constrained to the use of a single approach to handling uncertainty but should enable the combination and integration of different approaches in a way appropriate to each application.

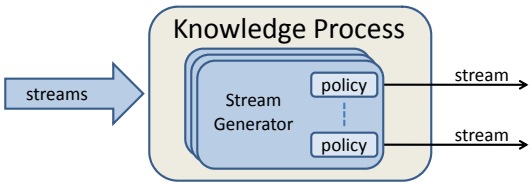


Figure 1: A prototypical knowledge process

Support flexible configuration and reconfiguration. When an agent's resources are insufficient, either due to lack of processing power or due to sensory limitations, various forms of trade-offs may be required. For example, update frequencies may be lowered, maximum permitted processing delays may be increased, resource-hungry algorithms may be dynamically replaced with more efficient but less accurate ones, or the agent may focus its attention on only the most important aspects of its current task. Reconfiguration may also be necessary when the current context changes.

Provide a declarative specification of the information being generated and the information processing functionalities available. An agent should be able to reason about trade-offs and reconfiguration without outside help, which requires introspective capabilities. Specifically, it must be possible to determine what information is currently being generated as well as the potential effects of a reconfiguration. The declarative specification should provide sufficient detail to allow the agent to make rational trade-off decisions.

DyKnow

DyKnow is a fully implemented stream-based knowledge processing middleware framework providing both conceptual and practical support for structuring a knowledge processing system as a set of streams and computations on streams. Input can be provided by a wide range of distributed information sources on many levels of abstraction, while output consists of streams representing for example objects, attributes, relations, and events.

Knowledge processing for a physical agent is fundamentally incremental in nature. Each part and functionality in the system, from sensing up 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, we have therefore chosen 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 a

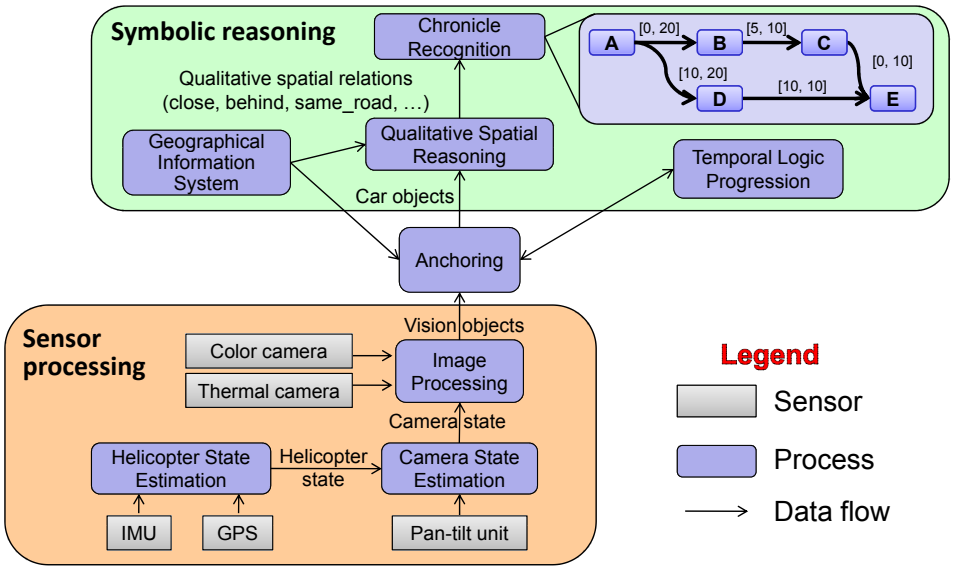


Figure 2: An overview of how the incremental processing required for a traffic surveillance task could be organized.

global 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 (Eugster *et al.* 2003), providing a solid foundation for distributed knowledge processing applications.

Each stream is associated with a *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 frequency, 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. For introspection purposes, policies are declaratively specified.

To summarize, a knowledge processing application in DyKnow consists of a set of knowledge processes connected by streams satisfying policies. An abstract view of a knowledge process is shown in Figure 1. Each knowledge process is an instantiation of a *source* or *computational unit* providing stream generators that generate streams. A source makes external information available in the form of streams while a computational unit refines and processes streams. A formal language called KPL is used to write declarative specifications of DyKnow applications (see (Heintz 2009; Heintz, Kvarnström, & Doherty 2010) for details). The DyKnow service, which implements the DyKnow framework, takes a set of KPL declarations and sets up the required processing and communication infrastructure. Due to the use of CORBA (Object Management Group 2008) for communication, knowledge processes are location-agnostic, providing support for distributed architectures running on multiple networked computers.

Fig. 2 provides an overview of how part of the incremental processing required for a traffic surveillance task could be organized as a set of distinct 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 UAV. 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.

Symbolic formalisms such as chronicle recognition (Ghallab 1996) require a consistent assignment of 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 to be described in the next section, which uses *progression* of formulas in a metric temporal logic to evaluate potential hypotheses about the observed objects. The anchoring system also assists in object classification and in the extraction of higher level attributes of an

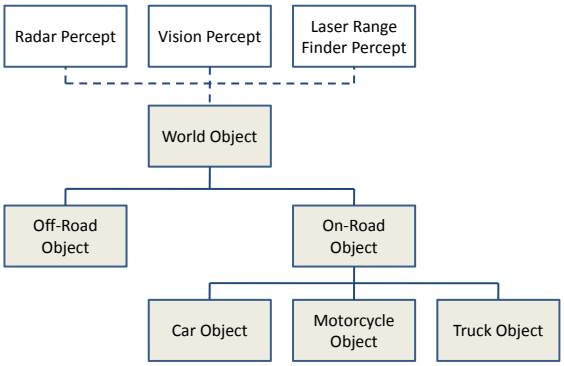


Figure 3: The example percept (white) / object (gray) hierarchy used in the traffic monitoring scenario.

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 beside(car_1, car_2) and 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.

Support for Anchoring

Most reasoning systems assume perfect knowledge about the identity of objects. For example, a planner assumes that all objects in its planning domain are distinct and unique. An important problem, especially when bridging the sense-reasoning gap, is therefore to detect objects in streams of sensor data and to reason about their identities. The problem of how to create and maintain a consistent correlation between symbolic representations of objects and sensor data that is being continually collected about these objects is called the *anchoring problem* (Coradeschi & Saffiotti 2003), which is a special case of the symbol grounding problem (Harnad 1990). A concrete example is to detect and track cars in a traffic monitoring application using a UAV equipped with color and thermal cameras.

Tracking an object, such as a car, through a series of images is a classical problem. There are many effective solutions for the case where the object is easily distinguishable and can be tracked without interruptions. However, we must also consider the case where an object is temporarily hidden by obstacles (or tunnels in the case of traffic), and where many similar objects may be present in the world. In this case, pure image-based tracking is not a complete solution, since it usually only considers the information available in the image itself. A more robust approach would need to actively reason about available knowledge of the world at higher abstraction levels, such as the normative characteristics of specific classes of physical objects. In the case of traffic, this would include the layout of the road network and the typical size, speed, and driving behavior of cars. It has been argued that anchoring can be seen as an extension to classical tracking approaches which handles missing data in a principled manner (Fritsch *et al.* 2003).

Existing approaches to anchoring work under the limiting assumption that each individual piece of sensor data, such as a blob found in a single frame from a video camera, should be anchored to a symbol in a single step. We believe that much can be gained in terms of accuracy as well as speed of recognition by taking advantage of the fact that one generally has access to a timed sequence, or stream, of sensor data related to a particular object.

We have therefore extended DyKnow with a stream-based hierarchical anchoring framework for incrementally anchoring symbols to streams of sensor data (Heintz, Kvarnström, & Doherty 2009). The anchoring process constructs and maintains a set of *object linkage structures* representing the best possible hypotheses at any time. The anchoring hypotheses are continually monitored and refined as more and more information becomes available. Symbols can be associated with an object at any level of classification, permitting symbolic reasoning on different levels of abstraction.

An example hierarchy for the traffic monitoring scenario can be seen in Fig. 3. A *world object* represents a physical object in the world. Its attributes are based on information from one or more linked percepts and include the absolute coordinates of the object in the physical world. World objects could either be *on-road objects* moving along roads or *off-road objects* not following roads. An on-road object has attributes representing the road segment or crossing the object occupies, making more qualitative forms of reasoning possible, and an improved position estimation which is snapped to the road. Finally, an on-road object could be a *car*, a *motorcycle*, or a *truck*. Each level in the hierarchy adds more abstract and qualitative information while still maintaining a copy of the attributes of the object it was derived from. Thus, an on-road object contains both the original position from the world object and the position projected onto the road network.

Hypotheses about object types and identities must be able to evolve over time. For example, while it might be determined quickly that a world object is an on-road object, more time may be required to determine that it is in fact a car. Also, what initially appeared to be a car might later turn out to be better classified as a truck. To support incremental addition of information and incremental revision of hypotheses, a single physical object is not represented as an indivisible object structure but as an *object linkage structure*.

An object linkage structure consists of a set of *objects* which are *linked* together (note that percepts are also considered to be objects). Each object has a type and is associated with a symbol, and represents information about a particular physical object at a given level of abstraction. A symbol is *anchored* if its object is part of an object linkage structure that is grounded in at least one percept. An example is shown in Fig. 4, representing the hypothesis that vision percept vp8, world object wo5, on-road object oo3, and car object co2 all correspond to the same physical object.

Whenever a new object of a given type is generated, it must be determined whether it also belongs to a particular subtype in the hierarchy. For example, a new vision per-

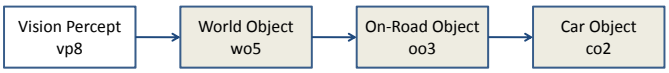


Figure 4: An example object linkage structure.

cept originating in image processing may be classified as corresponding to a world object. In this case, it must also be linked to a world object structure, thereby generating an object linkage structure. However, it is essential for anchoring that sensor data can be anchored even to symbols / objects for which no percepts have arrived for a period of time. Thus, objects and their symbols are not immediately removed when their associated percepts disappear, and any new object at one level might correspond to either a new object or an *existing* object at the next level. To reduce the computational cost, objects which are not likely to be found again are removed. Currently we discard objects which have not been observed or anchored for a certain application-dependent time.

Three conditions are used to determine when to add and remove links between objects belonging to types A and B. These conditions are written in an expressive temporal logic, similar to the well known Metric Temporal Logic (Koymans 1990), and incrementally evaluated by DyKnow using progression over a timed state sequence.[1]

The unary *establish condition* expresses when an object of type A, which may be a percept or a higher level object, should be linked to a *new* object of type B. When a new object of type A is created, the anchoring system immediately begins evaluating this condition. If and when the condition becomes true, a link to a new object of type B is created. Concrete examples will be given below. This corresponds to the Find functionality suggested by Coradeschi and Saffiotti (Coradeschi & Saffiotti 2003), which takes a symbolic description of an object and tries to anchor it in sensor data.

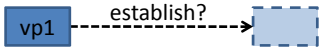
The binary *reestablish condition* expresses the condition for an object of type A to be linked to a *known* object of type B, as in the case where a new world object corresponds to an existing on-road object that had temporarily been hidden by a bridge. When a new object of type A is created, the anchoring system immediately begins to evaluate the reestablish condition for every known object of type B that is not linked to an object of type A. If and when one of these conditions becomes true, a link is created between the associated objects. This corresponds to the Reacquire functionality (Coradeschi & Saffiotti 2003).

While two objects are linked, the attributes of the more specific object are computed using a DyKnow computational unit from the attributes of the less specific object, possibly together with information from other sources.

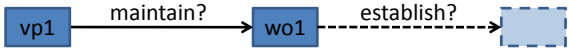
Finally, since observations are uncertain and classification is imperfect, any link created between two objects is considered a hypothesis and is continually validated through a *maintain condition*. Such conditions can compare the observed behavior of an object with behavior that is normative for its type, and possibly with behavior predicted in other ways. For example, one might state that an on-road object should remain continually on the road, maybe with occasional shorter periods being observed off the road due to sensor error.

If a maintain condition is violated, the corresponding link is removed. However, all objects involved remain, enabling re-classification and re-identification at a later time. The state of an object having no incoming links will be predicted based on a general model of how objects of this type normally behave. This corresponds to the Track functionality (Coradeschi & Saffiotti 2003).

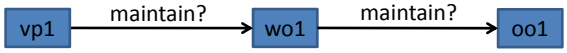
Example 1 *Assume that the image processing system is currently tracking a potential car represented by the vision percept vp1 and that no other objects have been created. Since there is a vision percept but no known world objects it is enough to evaluate the establish condition on vp1.*



If the establish condition is eventually satisfied, a new world object wo1 is created which is associated with vp1. As long as wo1 is associated with vp1 its state will be computed from the state of the vision percept vp1. It is also possible to estimate a model of the behavior of wo1 using the collected information. This model can later be used to predict the behavior of wo1 if it is no longer tracked. The maintain condition is monitored to continually verify the hypothesis that wo1 is a world object.

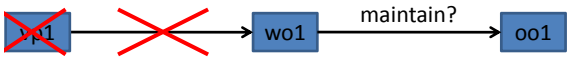


Further, assume that after a while wo1 has been hypothesized as being an on-road object represented by oo1 and an object linkage structure has been created containing all three objects. For example, one could assume that an object is an on-road object after it has been observed on a road for at least 30 seconds.

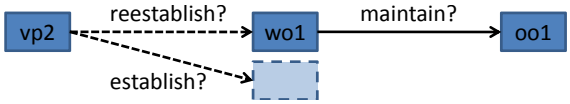


Assume the image processing system loses track of the potential car after a while. Then vp1 is removed together with the link to wo1. Even though the link is removed the world object wo1 remains as well as the on-road object oo1 and its link to wo1. While wo1 is not linked to any vision percept its state will be predicted using either a general model of physical objects or an individually adapted model of this particular object. Since wo1 is linked to an on-road object it is also possible to use this hypothesis to further restrict the predicted movements of the object as it can be assumed to only move along roads. This greatly reduces the possible positions of the object and makes it much easier to reestablish a link to it.

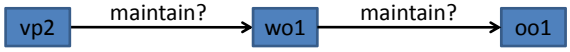
[1]Progression incrementally evaluates formulas in a state sequence. The result of progressing a formula through the first state in a sequence is a new formula that holds in the remainder of the state sequence iff the original formula holds in the complete state sequence. If progression returns true (false), the entire formula must be true (false), regardless of future states.



Assume further that the image processing system later recognizes a new potential car represented by the vision percept vp2. Since there exists a known world object, wo1, the knowledge process has to evaluate whether vp2 is a new world object, the known world object wo1, or not a world object at all. This is done by evaluating the establish condition on vp2 and the reestablish condition between vp2 and wo1.



Assume that after a while the establish condition is progressed to false and the (in this case unique) reestablish condition is progressed to true. Then a new link is created from vp2 to wo1 and the attributes of wo1 can be computed from the attributes of vp2. This also means that oo1 is once again anchored. To verify the new hypothesis a maintain condition between wo1 and vp2 is monitored.



□

Support for Planning

One approach to solving complex problems is to use a task planner to generate a plan that will achieve the goal. To integrate task planners into an embedded reasoning system there are a number of issues to consider.

Initial state. For a planner to be able to generate a plan which is relevant in the current situation it must have an accurate and up-to-date domain model. In a static environment it is possible to write a domain model once and for all since the world does not change. In a dynamic environment, such as a disaster area, we do not have the luxury of predefined static domain models. Instead, the UAV must itself generate information about the current state of the environment and encode this in a domain model.

Execution. Executing an action in a plan generally requires sophisticated feedback about the environment on different levels of abstraction. For example, a UAV following a three-dimensional trajectory must continually estimate its position by fusing data from several sensors, such as GPS and IMU. If it loses its GPS signal due to malfunction or jamming, vision-based landing may be needed, which requires processing video streams from cameras in order to estimate altitude and position relative to the landing site.

Monitoring. Classical task planners are built on the fundamental assumption that the only agent causing changes in the environment is the planner itself, or rather, the system or systems that will eventually execute the plan that it generates. Furthermore, they assume that all information provided to the planner as part of the initial state and the operator specifications is accurate. This may in some cases be a reasonable approximation of reality, but it is not always the case. Other agents might manipulate the environment of a system in ways that may prevent the successful execution of

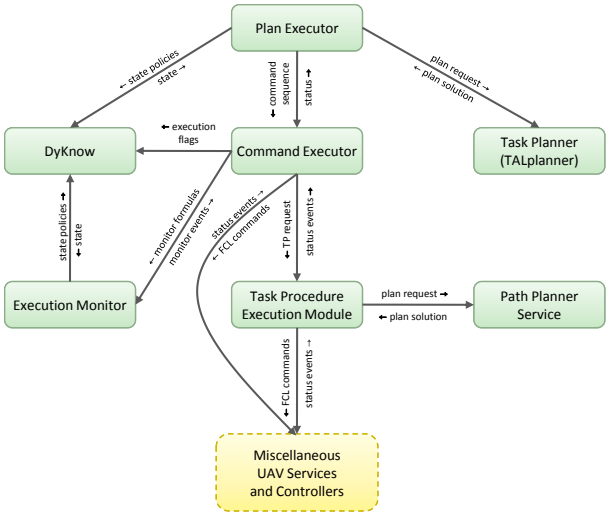


Figure 5: Task planning and execution monitoring overview

a plan. Sometimes actions can fail to have the effects that were modeled in a planning domain specification, regardless of the effort spent modeling all possible contingencies. Consequently, robust performance in a noisy environment requires some form of supervision, where the execution of a plan is constantly monitored in order to detect any discrepancies and recover from potential or actual failures.

We have developed a temporal logic-based task planning and execution monitoring framework which handles these issues and integrated it into our fully deployed rotor-based unmanned aircraft system (Doherty, Kvarnström, & Heintz 2009). In the spirit of cognitive robotics, we make specific use of Temporal Action Logic (TAL) (Doherty & Kvarnström 2008), a logic for reasoning about action and change. This logic has already been used as the semantic basis for a task planner called TALplanner (Doherty & Kvarnström 2001; Kvarnström 2005), which is used to generate mission plans that are carried out by an execution subsystem.

Knowledge gathered during plan execution can be used by DyKnow to incrementally create state structures. These state structures correspond to partial logical models in TAL, representing the actual development of the system and its environment over time. By specifying the desired behavior of the system and its environment using TAL formulas, violations of these formulas can be detected in a timely manner in an execution monitor subsystem, using a progression algorithm for prompt failure detection.

Figure 5 shows the relevant part of the UAV system architecture associated with task planning, execution of task plans and execution monitoring.

At the top of the center column is the *plan executor* which given a mission request calls the knowledge processing middleware DyKnow to acquire essential information about the current contextual state of the world or the UAV's own internal states. Together with a domain specification and a goal specification related to the current mission, this information is fed to TALplanner, which outputs a plan that will achieve

the designated goals, under the assumption that all actions succeed and no failures occur. Such a plan can also be automatically annotated with global and/or operator-specific conditions to be monitored during execution of the plan by an execution monitor in order to relax the assumption that no failures can occur. Such conditions are expressed as temporal logical formulas and evaluated on-line using formula progression techniques. The execution monitor notifies the command executor when actions do not achieve their desired results and one can then move into a plan repair phase.

Example 2 *Suppose that a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification:*

$$\Box \forall uav.(\mathit{power}(uav) > M \rightarrow \mathit{power} < f \cdot M \; U_{[0,\tau]} \; \Box_{[0,\tau']} \; \mathit{power}(uav) \leq M) \quad \Box$$

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

During plan execution, the command executor adds formulas to be monitored to the *execution monitor*. DyKnow continuously sends information about the development of the world in terms of state sequences to the monitor, which uses a progression algorithm to partially evaluate monitor formulas. The state sequences are generated from potentially asynchronous streams by a stream synchronization mechanism that uses the declarative policies of the input streams to determine when states should be created. If a violation is detected, this is immediately signaled as an event to the command executor, which can suspend the execution of the current plan, invoke an emergency brake command if required, optionally execute an initial recovery action, and finally signal new status to the plan executor. The plan executor is then responsible for completing the recovery procedure.

The fully integrated system is implemented on our UAVs. Plans are generated in the millisecond to seconds range using TALplanner and empirical testing shows that this approach is promising in terms of integrating high-level deliberative capability with lower-level reactive and control functionality.

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

Discussion

In the beginning of the paper we introduced a number of requirements for middleware for embedded reasoning. In this section we argue that DyKnow provides a significant degree of support for each of those requirements.

Support integration of existing reasoning functionality. Streams provide a powerful yet very general representation of information varying over time, and any reasoning functionalities whose inputs can be modeled as streams can easily be integrated using DyKnow. As two concrete examples, we have shown how progression of temporal logical formulas (Doherty, Kvarnström, & Heintz 2009) and chronicle recognition (Heintz, Rudol, & Doherty 2007) can be integrated using DyKnow.

Support distributed sources and processing. DyKnow satisfies this requirement through the use of the general concepts of streams and knowledge processes. Since the implementation is CORBA-based it provides good support for distributed applications. DyKnow explicitly represents both the time when information is valid and when it is available. Therefore it has excellent support for integrating information over time even with varying delays. DyKnow also provides a very useful stream synchronization mechanism that uses the declarative policies to determine how to synchronize a set of asynchronous streams and derive a stream of states. This functionality is for example used to create synchronized state sequences over which temporal logical formulas can be evaluated.

Support processing on many levels of abstraction. General support is provided in DyKnow through streams, where information can be sent at any abstraction level from raw sampled sensor data and upwards. The use of knowledge processes also provides general support for arbitrary forms of processing. At the same time, DyKnow is explicitly designed to be extensible to provide support for information structures and knowledge processing that is more specific than arbitrary streams. DyKnow provides direct support for specific forms of high-level information structures, such as object linkage structures, and specific forms of knowledge processing, including formula progression and chronicle recognition. This provides initial support for knowledge processing at higher levels than plain streams of data. In (Heintz & Doherty 2006) we argue that this support is enough to provide an appropriate framework for supporting all the functional abstraction levels in the JDL Data Fusion Model.

Support quantitative and qualitative processing. Streams provide support for arbitrarily complex data structures, from real values to images to object structures to qualitative relations. The structured content of samples also allows quantitative and qualitative information to be part of the same sample. DyKnow also has explicit support for combining qualitative and quantitative processing in the form of chronicle recognition, progression of metric temporal logical formulas, and object linkage structures. Both chronicles and temporal logical formulas support expressing conditions combining quantitative time and qualitative features.

Support bottom-up data processing and top-down model-based processing. Streams are directed but can be connected freely, giving the application programmer the possibility to do both top-down and bottom-up processing. Though this article has mostly used bottom-up processing, chronicle

recognition is a typical example of top-down model-based processing where the recognition engine may control the data being produced depending on the general event pattern it is attempting to detect.

Support management of uncertainty. In principle, DyKnow supports any approach to representing and managing uncertainty that can be handled by processes connected by streams. It is for example easy to add a probability or certainty factor to each sample in a stream. This information can then be used by knowledge processes subscribing to this stream. Additionally, DyKnow has explicit support for uncertainty in object identities and in the temporal uncertainty of complex events that can be expressed both in quantitative and qualitative terms. The use of a metric temporal logic also provides several ways to express temporal uncertainty.

Support flexible configuration and reconfiguration. Flexible configuration is provided by the declarative specification language KPL, which allows an application designer to describe the different processes in a knowledge processing application and how they are connected with streams satisfying specific policies. The DyKnow implementation uses the specification to instantiate and connect the required processes.

Provide a declarative specification of the information being generated and the information processing functionalities available. This requirement is satisfied through the formal language KPL for declarative specifications of DyKnow knowledge processing applications, as already described. The specification explicitly declares the properties of the streams by policies and how they connect the different knowledge processes.

Conclusion

Embedded reasoning is reasoning which is part of a larger system and whose purpose is to support or improve the system in some way. This has for example the consequence that the reasoning must be integrated in and interact with the larger system. Middleware for embedded reasoning is software which supports and simplifies this integration and interaction. In this paper we have given a high-level overview of the stream-based middleware DyKnow and how it can support this type of embedded reasoning. More specifically we discussed general requirements on middleware and how DyKnow can be used to integrate embedded reasoning in the form of anchoring and planning in an autonomous system.

Our conclusion is that middleware for embedded reasoning should support declarative specifications for flexible configuration and dynamic reconfiguration of distributed context dependent processing at many different levels of abstraction and that DyKnow provides suitable support for integrating several types of embedded reasoning. The conclusion is based on the integration of for example planning, execution monitoring, and chronicle recognition in sophisticated autonomous UAV applications. However, this is only the first step towards developing robust and general middleware support for embedded reasoning.

References

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

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

Doherty, P., and Kvarnström, J. 2008. Temporal action logics. In Lifschitz, V.; van Harmelen, F.; and Porter, F., eds., *Handbook of Knowledge Representation*. Elsevier.

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 *Proc. of DARS*.

Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. 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.

Doherty, P. 2004. Advanced research with autonomous unmanned aerial vehicles. In *Proc. of KR*.

Eugster, P. T.; Felber, P. A.; Guerraoui, R.; and Kermarrec, A.-M. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35(2):114–131.

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

Ghallab, M. 1996. On chronicles: Representation, on-line recognition and learning. In *Proc. of KR*.

Harnad, S. 1990. The symbol-grounding problem. *Physica* D(42):335–346.

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

Heintz, F., and Doherty, P. 2006. A knowledge processing middleware framework and its relation to the JDL data fusion model. *Journal of Intelligent and Fuzzy Systems* 17(4).

Heintz, F.; Kvarnström, J.; and Doherty, P. 2009. A stream-based hierarchical anchoring framework. In *Proc. of IROS*.

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

Heintz, F.; Rudol, P.; and Doherty, P. 2007. From images to traffic behavior – a UAV tracking and monitoring application. In *Proc. of Fusion*.

Heintz, F. 2009. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework*. Ph.D. Dissertation, Linköpings universitet.

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

Kvarnström, J. 2005. *TALplanner and Other Extensions to Temporal Action Logic*. Ph.D. Dissertation, Linköpings universitet.

Object Management Group. 2008. The CORBA specification v 3.1.