Knowledge Processing Middleware

Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty
{frehe, jonkv, patdo}@ida.liu.se

Dept. of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden

Abstract. Developing autonomous agents displaying rational and goal-directed behavior in a dynamic physical environment requires the integration of a great number of separate deliberative and reactive functionalities. This integration must be built on top of a solid foundation of data, information and knowledge having numerous origins, including quantitative sensors and qualitative knowledge databases. Processing is generally required on many levels of abstraction and includes refinement and fusion of noisy sensor data and symbolic reasoning. We propose the use of *knowledge processing middleware* as a systematic approach for organizing such processing. Desirable properties of such middleware are presented and motivated. We then argue that a declarative stream-based system is appropriate to provide the desired functionality. Different types of knowledge processes and components of the middleware are described and motivated in the context of a UAV traffic monitoring application. Finally DyKnow, a concrete example of stream-based knowledge processing middleware, is briefly described.[1]

1 Introduction

When developing autonomous agents displaying rational and goal-directed behavior in a dynamic physical environment, we can lean back on decades of research in artificial intelligence. A great number of deliberative and reactive functionalities have already been developed, including chronicle recognition, motion planning, task planning and execution monitoring. Integrating these approaches into a coherent system requires reconciling the different formalisms they use to represent information and knowledge about the world. To construct these world models and maintain a correlation between them and the environment, information and knowledge must be extracted from data collected by sensors. However, most research done in a symbolic context tends to assume crisp knowledge about the current state of the world while information extracted from the environment often consists of noisy and incomplete quantitative data on a much lower level of abstraction. This causes a wide gap between sensing and reasoning.

Bridging this gap in a single step, using a single technique, is only possible for the simplest of autonomous systems. As complexity increases, one typically requires a combination of a wide variety of methods, including more or less standard functionalities such as various forms of image processing and information fusion as well as

[1] This work is partially supported by grants from the Swedish Aeronautics Research Council (NFFP4-S4203), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII and the Center for Industrial Information Technology CENIIT (06.09).

application-specific and possibly even scenario-specific approaches. Such integration is often done ad hoc using a variety of mechanisms within a single architecture, partly by allowing the sensory and deliberative layers of a system to gradually extend towards each other and partly by introducing intermediate processing levels.

We propose using the term *knowledge processing middleware* for a principled and systematic framework for organizing incremental and potentially distributed processing of knowledge at many levels of abstraction. Rather than being a robotic architecture itself, knowledge processing middleware should provide an infrastructure for integrating the necessary components in such an architecture and managing the information flow between these components. It should support incremental processing of sensor data and facilitate generating a coherent view of the environment at increasing abstraction levels, eventually providing knowledge at a level natural to use in symbolic deliberative functionalities. It should also support the integration of different deliberation techniques.

In the next section, an example scenario is presented as further motivation for the need for a systematic knowledge processing middleware framework. Desirable properties of such frameworks are investigated and a specific stream-based architecture suitable for a wide range of systems is proposed. As a concrete example, our framework DyKnow is briefly described. We conclude with some related work and a summary.

2 A Traffic Monitoring Scenario

Traffic monitoring is an important application domain for autonomous unmanned aerial vehicles (UAVs), where tasks such as detecting accidents and traffic violations and finding accessible routes for emergency vehicles provide a plethora of cases demonstrating the need for an intermediary layer between sensing and deliberation.

One approach to detecting traffic violations uses a formal declarative description of each type of violation. This can be done using a chronicle [1], which defines a class of complex events using a simple temporal network where nodes correspond to occurrences of high level qualitative events and edges correspond to metric temporal constraints. For example, to detect a reckless overtake, events corresponding to changes in qualitative spatial relations such as $\mathit{beside}(\mathit{car}_1, \mathit{car}_2)$ and $\mathit{on}(\mathit{car}, \mathit{road})$ might be used. Creating such representations from low-level sensory data, such as video streams, involves a great deal of work at different levels of abstraction which would benefit from being separated into distinct and systematically organized tasks. Figure 1 provides an overview of how this processing could be organized. We emphasize that this is intended to illustrate one potential use for knowledge processing middleware rather than to propose a specific robotic architecture to be used in UAV applications.

At the lowest level, a helicopter state estimator uses data from an inertial measurement unit (IMU) and a GPS sensor to determine the current position and attitude of the UAV. This information is fed into a camera state estimator, together with the current angles of the pan-tilt unit on which color and infrared cameras are mounted, to determine the current camera state. The image processing system uses the camera state to determine where the cameras are currently pointing. The two video streams can then be analyzed in order to extract vision objects representing hypotheses regarding moving and stationary physical entities, including their approximate positions and velocities.

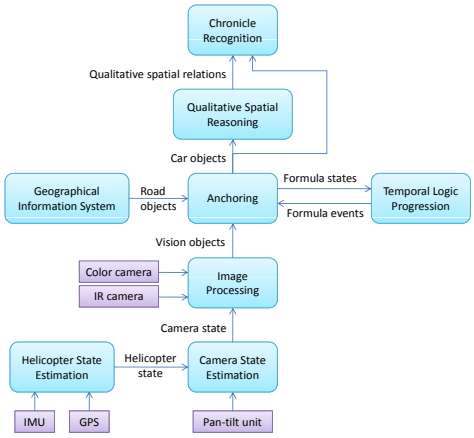

Fig. 1. Incremental Processing

Each vision object must be associated with a symbol for use in higher level services, a process known as *anchoring* [2, 3]. Identifying which vision objects correspond to vehicles is also essential, which requires knowledge about normative sizes and behaviors of vehicles. Behaviors can be described using formulas in a metric temporal modal logic, which are incrementally progressed through states that include current vehicle positions, velocities, and other relevant information. An entity satisfying all requirements can be hypothesized to be a vehicle, a hypothesis that may be withdrawn if the progressor signals that the entity has ceased to satisfy the normative behavior.

As an example, vehicles usually travel on roads. Given that image processing provides absolute world coordinates for each vision object, the anchoring process can query a geographic information system to determine the nearest road segment and derive higher level predicates such as *on-road*(car) and *in-crossing*(car). These would be included in the states sent to the progressor as well as in the vehicle objects sent to the next stage of processing, which involves deriving qualitative spatial relations between vehicles such as *beside*(car_1, car_2) and *close*(car_1, car_2). These predicates, and the concrete events corresponding to changes in the predicates, finally provide sufficient information for the chronicle recognition system to determine when higher-level events such as reckless overtakes occur.

In this example, a considerable number of distinct processes are involved in bridging the gap between sensing and deliberation and generating the necessary symbolic representations from sensor data. However, to fully appreciate the complexity of the system, we have to widen our perspective. Towards the smaller end of the scale, we can see that a single process in Figure 1 is sometimes merely an abstraction of what is in fact a set of distinct processes. Anchoring is a prime example, encapsulating tasks such as the derivation of higher level predicates which could also be viewed as a separate process. At the other end of the scale, a complete UAV system also involves numerous other sensors and information sources as well as services with distinct knowledge requirements, including task planning, path planning, execution monitoring, and reactive procedures.

Consequently, what is seen in Figure 1 is merely an abstraction of the full complexity of a small part of the system. It is clear that a systematic means for integrating all forms of knowledge processing, and handling the necessary communication between parts of the system, would be of great benefit. Knowledge processing middleware should fill this role, by providing a standard framework and infrastructure for integrating image processing, sensor fusion, and other data, information and knowledge processing functionalities into a coherent system.

3 Knowledge Processing Middleware

As stated in the introduction, any form of knowledge processing middleware should provide a principled and systematic framework for bridging the gap between sensing and deliberation in a physical agent. While it is unlikely that one will ever achieve universal agreement on the detailed requirements for such middleware, the following requirements have served as important guiding principles.

First, the framework should *permit the integration of information from distributed sources*, allowing this information to be processed at many different levels of abstraction and transformed into a suitable form for use by a deliberative functionality. In traffic monitoring, the primary input will consist of low level sensor data such as images, a signal from a barometric pressure sensor, a GPS signal, laser range scans, and so on. There might also be high level information available such as geographical information and declarative specifications of traffic patterns and normative behaviors of vehicles. The middleware must be sufficiently flexible to allow the integration of these sources into a coherent processing system. Since the appropriate structure will vary between applications, a general framework should be agnostic as to the types of data and information being handled and should not be limited to specific connection topologies.

Many applications, including traffic monitoring, provide a natural abstraction hierarchy starting with quantitative sensor signals, 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. Therefore a second requirement is the *support of quantitative and qualitative processing* as well as a mix of them.

A third requirement is that *both bottom-up data processing and top-down model-based processing should be supported*. Different abstraction levels are not independent. Each level is dependent on the levels below it to get input for bottom-up data processing. At the same time, the output from higher levels could be used to guide processing in a top-down fashion. For example, if a vehicle is detected on a particular road segment, then a vehicle model could be used to predict possible future locations, which could be used to direct or constrain the processing on lower levels.

A fourth requirement is support for *management of uncertainty*. Many types of uncertainty exist, at the quantitative sensor data level as well as in the symbolic identity of objects and in temporal and spatial aspects of events and situations. It should be possible to use different approaches in different architectures implemented with knowledge processing middleware, and to integrate multiple approaches in a single application.

Physical agents acting in the world have limited sensory capabilities and limited resources. At times these resources may be insufficient for satisfying all currently executing tasks, and trade-offs may be necessary. For example, reducing update frequencies would cause less information to be generated, while increasing the maximum permitted processing delay would provide more time to complete processing. Similarly, an agent might decide to focus its attention on the most important aspects of its current situation, ignoring events or objects in the periphery, or to focus on providing information for the highest priority tasks or goals. Resource-hungry calculations can sometimes be replaced with more efficient but less accurate ones. Each trade-off will have effects on the quality of the information produced and the resources used. A fifth requirement on

knowledge processing middleware is therefore support for *flexible configuration and reconfiguration*. This is also necessary for context-dependent processing. For example, one may initially assume that vehicles follow roads. If a vehicle goes off road, this simplifying assumption must be retracted and processing may need to be reconfigured.

It should be possible to provide an agent implemented using knowledge processing middleware with the ability to reason about trade-offs and reconfigure itself without outside help, which requires introspective capabilities. Specifically, the agent must be able to determine what information is currently being generated as well as the potential effects of any changes it may make in the processing structure. Therefore a sixth requirement is for the framework to provide a *declarative specification of the information being generated and the processing functionalities that are available*, with sufficient content to make rational trade-off decisions.

To summarize, we believe knowledge processing middleware should support declarative specifications for flexible configuration and dynamic reconfiguration of context dependent processing at many different levels of abstraction.

4 Stream-Based Knowledge Processing Middleware

The previous section focused on a set of requirements, intentionally leaving open the question of how these requirements should be satisfied. We now go on to propose *stream-based* knowledge processing middleware, one specific type of framework which we believe will be useful in many applications. A concrete implementation, DyKnow, will be discussed later in this paper.

Due to the need for incremental refinement of information at different levels of abstraction, we model computations and processes within the stream-based knowledge processing framework as active and sustained *knowledge processes*. The complexity of such processes may vary greatly, ranging from simple adaptation of raw sensor data to image processing algorithms and potentially reactive and deliberative processes.

In our experience, it is not uncommon for knowledge processes at a lower level to require information at a higher frequency than those at a higher level. For example, a sensor interface process may query a sensor at a high rate in order to average out noise, providing refined results at a lower effective sample rate. This requires knowledge processes to be decoupled and asynchronous to a certain degree. In stream-based knowledge processing middleware, this is achieved by allowing a knowledge process to declare a set of *stream generators*, each of which can be *subscribed* to by an arbitrary number of processes. A subscription can be viewed as a continuous query, which creates a distinct asynchronous *stream* onto which new data is pushed as it is generated. The contents of a stream may be seen by the receiver as data, information or knowledge.

Decoupling processes through asynchronous streams minimizes the risk of losing samples or missing events, something which can be a cause of problems in query-based systems where it is the responsibility of the receiver to poll at sufficiently high frequencies. Streams can provide the necessary input for processes that require a constant and timely flow of information. For example, a chronicle recognition system needs to be apprised of all pertinent events as they occur, and an execution monitor must receive constant updates for the current system state at a given minimum rate. A push-based

stream system also lends itself easily to "on-availability" processing, i.e. processing data as soon as it is available, and the minimization of processing delays, compared to a query-based system where polling introduces unnecessary delays in processing and the risk of missing potentially essential updates as well as wastes resources. Finally, decoupling also facilitates the distribution of processes within a platform or between different platforms, another important property of many complex autonomous systems.

Finding the correct stream generator requires each stream generator to have an identity which can be referred to, a *label*. Though a label could be opaque, it often makes sense to use structured labels. For example, given that there is a separate position estimator for each vehicle, it makes sense to provide an identifier i for each vehicle and to denote the (single) stream generator of each position estimator by $\mathit{position}[i]$. Knowing the vehicle identifier is sufficient for generating the correct stream generator label.

Even if many processes connect to the same stream generator, they may have different requirements for their input. As an example, one could state whether new information should be sent "when available", which is reasonable for more event-like information or discrete transitions, or with a given frequency, which is more reasonable with continuously varying data. In the latter case, a process being asked for a subscription at a high frequency may need to alter its own subscriptions to be able to generate stream content at the desired rate. Requirements may also include the desired approximation strategy when the source knowledge process lacks input, such as interpolation or extrapolation strategies or assuming the previous value persists. Thus, every subscription request should include a *policy* describing such requirements. The stream is then assumed to satisfy this policy until it is removed or altered. For introspection purposes, policies should be declaratively specified.

While it should be noted that not all processing is based on continuous updates, neither is a stream-based framework limited to being used in this manner. For example, a path planner or task planner may require an initial state from which planning should begin, and usually cannot take updates into account. Even in this situation, decoupling and asynchronicity are important, as is the ability for lower level processing to build on a continuous stream of input before it can generate the desired snapshot. A snapshot query, then, is simply a special case of the ordinary continuous query.

4.1 Knowledge Processes

For the purpose of modeling, we find it useful to identify four distinct types of knowledge process: Primitive processes, refinement processes, configuration processes and mediation processes.

Primitive processes serve as an interface to the outside world, connecting to sensors, databases or other information sources that in themselves have no explicit support for stream-based knowledge processing. Such processes have no stream inputs but provide a non-empty set of stream generators. In general, they tend to be quite simple, mainly adapting data in a multitude of external representations to the stream-based framework. For example, one process may use a hardware interface to read a barometric pressure sensor and provide a stream generator for this information. However, greater complexity is also possible, with primitive processes performing tasks such as image processing.

The remaining process types will be introduced by means of an illustrating example

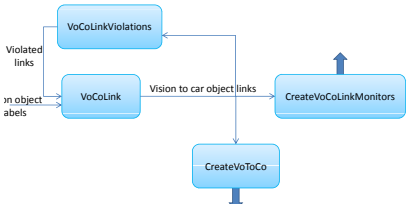

Fig. 2. Before creating vision object

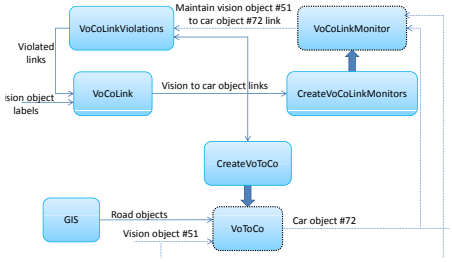

Fig. 3. VisionObject#51 linked to CarObject#72.

from the traffic monitoring scenario, where car objects must be generated and anchored to sensor data collected using cameras. This example shows one of many potential solutions that can be implemented with the help of knowledge processing middleware and has been successfully used in test flights with an experimental UAV platform [4].

In the implemented approach, the image processing system produces *vision objects* representing entities found in an image, having visual and thermal properties similar to those of a car. A vision object state contains an estimation of the size of the entity and its position in absolute world coordinates. When a new vision object has been found, it is tracked for as long as possible by the image processing system and each time it is found in an image a new vision object state is pushed on a stream.

Anchoring begins with this stream of vision object states, aiming at the generation of a stream of *car object* states providing a more qualitative representation, including relations between car objects and road segments. An initial filtering process, omitted here for brevity, determines whether to hypothesize that a certain vision object in fact corresponds to a car. If so, a car object is created and a *link* is established between the two objects. To monitor that the car object actually behaves like a car, a maintenance constraint describing expected behavior is defined. The constraint is monitored, and if violated, the car hypothesis is withdrawn and the link is removed. A temporal modal logic is used for encoding normative behaviors, and a progression algorithm is used for monitoring that the formula is not violated.

Figure 2 shows an initial process setup, existing when no vision objects have been linked to car objects. As will be seen, processes can dynamically generate new processes when necessary. Figure 3 illustrates the process configuration when VisionObject#51 has been linked to CarObject#72 and two new refinement processes have been created.

The first process type to be considered is the *refinement process*, which takes a set of streams as input and provides one or more stream generators producing refined, abstracted or otherwise processed values. Several examples can be found in the traffic monitoring application, such as:

– VoCoLink – Manages the set of links between vision objects and car objects, each link being represented as a pair of labels. When a previously unseen vision object label is received, create a new car object label and a link between them. When a link is received from the VoCoLinkViolations process, the maintenance constraint of the link has been violated and the link is removed. The output is a stream of sets of

links. A suitable policy may request notification only when the set of links changes.

- VoToCo – Refines a single vision object to a car object by adding qualitative information such as which road segment the object is on and whether the road segment is a crossing or a road. Because quantitative data is still present in a car object, a suitable policy may request new information to be sent with a fixed sample frequency. Using a separate process for each car object yields a fine-grained processing network where different cars may be processed at different frequencies depending on the current focus of attention.
- VoCoLinkMonitor – An instantiation of the formula progressor. Monitors the maintenance constraint of a vision object to car object link, using the stream of car object states generated by the associated VoToCo. The output is false iff the maintenance constraint has been violated.

The second type of process, the *configuration process*, takes a set of streams as input but produces no new streams. Instead, it enables dynamic reconfiguration by adding or removing streams and processes. The configuration processes used in our example are:

- CreateVoCoLinkMonitors – Takes a stream of sets of links and ensures VoCoLinkMonitor refinement processes are created and removed as necessary.
- CreateVoToCos – Takes a stream of vision to car object links and ensures VoToCo refinement processes are created and removed as necessary.

Finally, a *mediation process* generates streams by selecting or collecting information from other streams. Here, one or more of the inputs can be a stream of labels identifying other streams to which the mediation process may subscribe. This allows a different type of dynamic reconfiguration in the case where not all potential inputs to a process are known in advance or where one does not want to simultaneously subscribe to all potential inputs due to processing cost. One mediation process is used in our example:

- VoCoLinkViolations – Takes a stream of sets of links identifying all current connections between vision objects and car objects. Dynamically subscribes to and unsubscribes from monitor information from the associated VoCoLinkMonitors as necessary. If a monitor signals a violation (sending the value "false"), the corresponding link becomes part of the output, a stream of sets of violated links.

In Figure 2 the VoCoLinkViolations mediation process subscribes to no streams, since there are no VoCoLinkMonitor streams. In Figure 3 it subscribes to the stream of monitor results of the maintenance constraint of the new VisionObject#51 to CarObject#72 link.

This example shows how stream-based knowledge processing middleware can be applied in a very fine-grained manner, even at the level of individual objects being tracked in an image processing context. At a higher level, the entire anchoring process can be viewed as a composite knowledge process with a small number of inputs and outputs, as originally visualized in Figure 1. Thus, one can switch between different abstraction levels while remaining within the same unifying framework. In previous work it has been shown how stream-based knowledge processing middleware can provide support for the different functional levels in the JDL Data Fusion Model [5].

4.2 Timing

Any realistic knowledge processing architecture must take into account the fact that both processing and communication takes time, and that delays may vary, especially in a distributed setting. As an example, suppose one knowledge process is responsible for determining whether two cars are too close to each other. This test could be performed by subscribing to two car position streams and measuring the distance between the cars every time a new position sample arrives. Should one input stream be delayed by one sample period, distance calculations would be off by the distance traveled during that period, possibly triggering a false alarm. Thus, the fact that two pieces of information arrive simultaneously must not be taken to mean that they refer to the same time.

For this reason, stream-based knowledge processing middleware should support tagging each piece of information in a stream with its *valid time*, the time at which the information was valid in the physical environment. For example, an image taken at time t has the valid time t. If an image processing system extracts vision objects from this image, each created vision object should have the same valid time even though some time will have passed during processing. One can then ensure that only samples with the same valid time are compared. Valid time is also used in temporal databases [6].

Note that nothing prevents the creation of multiple samples with the same valid time. For example, a knowledge process could very quickly provide a first rough estimate of some property, after which it would run a more complex algorithm and eventually provide a better estimate with identical valid time.

The *available time*, the time when a piece of information became available through a stream, is also relevant. If each value is tagged with its available time, a knowledge process can easily determine the total aggregated processing and communication delay associated with the value, which is useful in dynamic reconfiguration. Note that the available time is not the same as the time when the value was retrieved from the stream, as retrieval may be delayed by other processing.

The available time is also essential when determining whether a system behaves according to specification, which depends on the information actually available at any time as opposed to information that has not yet arrived.

5 DyKnow

A concrete example of a stream-based knowledge processing middleware framework called DyKnow has been developed as part of our effort to build UAVs capable of carrying out complex missions [5, 7, 8]. Most of the functionality provided by DyKnow has already been presented in the previous section, but one important decision for each concrete instantiation is the type of entities it can process. For modeling purposes, DyKnow views the world as consisting of *objects* and *features*.

Since we are interested in dynamic worlds, a feature may change values over time. To model the dynamic nature of the value of a feature a *fluent* is introduced. A fluent is a total function from time to value, representing the value of a feature at every time-point. Example features are the speed of a car, the distance between two cars, and the number of cars in the world.

Since the world is continuous and the sensors are imperfect the exact fluent of a feature will in most cases never be completely known, instead it has to be approximated. In DyKnow, an approximation of the value of a feature over time is represented by a *fluent stream*. A fluent stream is a totally ordered sequence of *samples*, where each sample represents an observation or an estimation of the value of the feature at a particular time-point.

To satisfy the sixth requirement of having a declarative specification of the information being generated, DyKnow introduces a formal language to describe knowledge processing applications. An application declaration describes what knowledge processes and streams exists and the constraints on them. To model the processing of a dependent knowledge process a *computational unit* is introduced. A computational unit takes one or more samples as inputs and computes zero or more samples as output. A computational unit is used by a dependent knowledge process to create a new fluent generator. A *fluent generator declaration* is used to specify the fluent generators of a knowledge process. It can either be primitive or dependent. To specify a stream a *policy* is used.

The DyKnow implementation sets up stream processing according to an application specification and processes streams to satisfy their policies. Using DyKnow an instance of the traffic monitoring scenario has successfully been implemented and tested [4].

6 Related Work

There is a large body of work on hybrid architectures which integrate reactive and deliberative decision making [9–13]. This work has mainly focused on integrating actions on different levels of abstraction, from control laws to reactive behaviors to deliberative planning. It is often mentioned that there is a parallel hierarchy of more and more abstract information extraction processes or that the deliberative layer uses symbolic knowledge, but only a few of these approaches are described in some detail [14–16].

We now focus on some approaches providing general support for integrating sensing and reasoning as opposed to approaches tackling important but particular subproblems such as symbol grounding, simultaneous localization and mapping, or transforming signals to symbols. With general support we mean that a system explicitly supports at least a few of the requirements, and does not prevent any of the remaining requirements from being met. However, the explicit support for the requirements often widely differ.

4D/RCS is a general cognitive architecture which can be used to combine different knowledge representation techniques [17]. It consists of a multi-layered hierarchy of computational nodes each containing sensory processing, world modeling, value judgment, behavior generation, and a knowledge database. The idea of the design is that the lowest levels have short-range and high-resolution representations of space and time appropriate for the sensor level while higher levels have long-range and low-resolution representations appropriate for deliberative services. Each level thus provides an abstract view of the previous levels. Each node may use its own knowledge representation and thereby support multiple different representation techniques. However, the architecture does not, to our knowledge, explicitly address the issues related to connecting different representations and transforming one representation into another. These are fundamental issues which stream-based knowledge processing middleware explicitly

supports. However, it ought to be possible to combine the two approaches and implement the 4D/RCS architecture using DyKnow.

The CoSy Architecture Schema Toolkit (CAST) built on top of the Boxes and Lines Toolkit (BALT) is a tool for creating cognitive architectures [18]. An architecture consists of a collection of interconnected subarchitectures (SAs). Each SA contains a set of processing components that can be connected to sensors and effectors and a working memory which acts like a blackboard within the SA. A processing component can either be managed or unmanaged. An *unmanaged* processing component runs constantly and directly pushes its results into the working memory. A *managed* process, on the other hand, monitors the working memory content for changes and suggests new possible processing tasks. Since these tasks might be computationally expensive a *task manager* uses a set of rules to decide which task should be executed next based on the current goals of the SA. One special SA is the *binder* which creates a high-level shared representation that relates back to low-level subsystem-specific representations [19]. It binds together content from separate information processing subsystems to provide symbols that can be used for deliberation and action.

The BALT middleware provides a set of processes which can be connected either by 1-to-1 *pull* connections or 1-to-N *push* connections. With its push connections and its support for distributing information and integrating reasoning components it can be seen as a basic stream-based knowledge processing middleware. A difference is that it does not provide any declarative policy-like specification to control push connections. CAST further adds support for a structured way of processing data on many levels of abstraction and the binder supports an explicit integration of representations from several SAs. A difference compared to DyKnow is the lack of a declarative specification of the processing of an architecture.

7 Summary

As autonomous physical systems become more sophisticated and are expected to handle increasingly complex and challenging tasks and missions, there is a growing need to integrate a variety of functionalities developed in the field of artificial intelligence. A great deal of research in this field has been performed in a purely symbolic setting, where one assumes the necessary knowledge is already available in a suitable high-level representation. There is a wide gap between such representations and the noisy sensor data provided by a physical platform, a gap that must somehow be bridged in order to ground the symbols that the system reasons about in the physical environment in which the system should act.

When physical autonomous systems grow in scope and complexity, bridging the gap in an ad-hoc manner becomes impractical and inefficient. At the same time, a systematic solution has to be sufficiently flexible to accommodate a wide range of components with highly varying demands. Therefore, we began by discussing the requirements that we believe should be placed on any principled approach to bridging the gap. As the next step, we proposed a specific class of approaches, which we call stream-based knowledge processing middleware and which is appropriate for a large class of autonomous systems. This step provides a considerable amount of structure for the integration of

the necessary functionalities, but still leaves certain decisions open in order to avoid unnecessarily limiting the class of systems to which it is applicable. Finally, DyKnow was presented to give an example of an existing implementation of such middleware.

References

1. Ghallab, M.: On chronicles: Representation, on-line recognition and learning. In: Proc. KR'96. (1996) 597–607
2. Coradeschi, S., Saffiotti, A.: An introduction to the anchoring problem. Robotics and Autonomous Systems **43**(2-3) (2003) 85–96
3. Heintz, F., Doherty, P.: Managing dynamic object structures using hypothesis generation and validation. In: Proc. Workshop on Anchoring Symbols to Sensor Data. (2004)
4. Heintz, F., Rudol, P., Doherty, P.: From images to traffic behavior – a UAV tracking and monitoring application. In: Proc. Fusion'07, Quebec, Canada (2007)
5. Heintz, F., Doherty, P.: A knowledge processing middleware framework and its relation to the JDL data fusion model. J. Intelligent and Fuzzy Systems **17**(4) (2006)
6. Jensen, C., Dyreson, C., eds.: The consensus glossary of temporal database concepts - february 1998 version. In: Temporal Databases: Research and Practice. (1998)
7. Doherty, P., Haslum, P., Heintz, F., Merz, T., Nyblom, P., Persson, T., Wingman, B.: A distributed architecture for autonomous unmanned aerial vehicle experimentation. In: Proc. DARS'04. (2004)
8. Heintz, F., Doherty, P.: DyKnow: An approach to middleware for knowledge processing. J. Intelligent and Fuzzy Systems **15**(1) (2004) 3–13
9. Bonasso, P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., Slack, M.: Experiences with an architecture for intelligent, reactive agents. J. Experimental and Theoretical AI **9** (1997)
10. Arkin, R.C.: Behavior-Based Robotics. MIT Press (1998)
11. Pell, B., Gamble, E.B., Gat, E., Keesing, R., Kurien, J., Millar, W., Nayak, P.P., Plaunt, C., Williams, B.C.: A hybrid procedural/deductive executive for autonomous spacecraft. In: Proc. AGENTS '98. (1998) 369–376
12. Atkin, M.S., King, G.W., Westbrook, D.L., Heeringa, B., Cohen, P.R.: Hierarchical agent control: a framework for defining agent behavior. In: Proc. AGENTS '01. (2001) 425–432
13. Scheutz, M., Kramer, J.: RADIC – a generic component for the integration of existing reactive and deliberative layers for autonomous robots. In: Proc. AAMAS'06. (2006)
14. Lyons, D., Arbib, M.: A formal model of computation for sensory-based robotics. Robotics and Automation, IEEE Transactions on **5**(3) (1989) 280–293
15. Konolige, K., Myers, K., Ruspini, E., Saffiotti, A.: The Saphira architecture: a design for autonomy. J. Experimental and Theoretical AI **9**(2–3) (1997) 215–235
16. Andronache, V., Scheutz, M.: APOC - a framework for complex agents. In: Proceedings of the AAAI Spring Symposium, AAAI Press (2003) 18–25
17. Schlenoff, C., Albus, J., Messina, E., Barbera, A.J., Madhavan, R., Balakrisky, S.: Using 4D/RCS to address AI knowledge integration. AI Mag. **27**(2) (2006) 71–82
18. Hawes, N., Zillich, M., Wyatt, J.: BALT & CAST: Middleware for cognitive robotics. In: Proceedings of IEEE RO-MAN 2007. (2007) 998–1003
19. Jacobsson, H., Hawes, N., Kruijff, G.J., Wyatt, J.: Crossmodal content binding in information-processing architectures. In: Proc. HRI'08, Amsterdam, The Netherlands (2008)