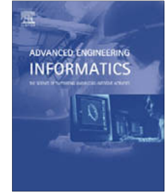
Advanced Engineering Informatics 24 (2010) 14–26

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei



Bridging the sense-reasoning gap: DyKnow – Stream-based middleware for knowledge processing ☆

Fredrik Heintz *, Jonas Kvarnström, Patrick Doherty

Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden

ARTICLE INFO

Article history:
Received 20 June 2009
Accepted 10 August 2009
Available online 12 October 2009

ABSTRACT

Engineering autonomous agents that display rational and goal-directed behavior in dynamic physical environments requires a steady flow of information from sensors to high-level reasoning components. However, while sensors tend to generate noisy and incomplete quantitative data, reasoning often requires crisp symbolic knowledge. The gap between sensing and reasoning is quite wide, and cannot in general be bridged in a single step. Instead, this task requires a more general approach to integrating and organizing multiple forms of information and knowledge processing on different levels of abstraction in a structured and principled manner.

We propose *knowledge processing middleware* as a systematic approach to organizing such processing. Desirable properties are presented and motivated. We argue that a declarative stream-based system is appropriate for the required functionality and present DyKnow, a concrete implemented instantiation of stream-based knowledge processing middleware with a formal semantics. Several types of knowledge processes are defined and motivated in the context of a UAV traffic monitoring application.

In the implemented application, DyKnow is used to incrementally bridge the sense-reasoning gap and generate partial logical models of the environment over which metric temporal logical formulas are evaluated. Using such formulas, hypotheses are formed and validated about the type of vehicles being observed. DyKnow is also used to generate event streams representing for example changes in qualitative spatial relations, which are used to detect traffic violations expressed as declarative chronicles.

© 2009 Elsevier Ltd. All rights reserved.

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 reasoning functionalities have already been developed, including chronicle recognition [1], motion planning [2,3], and task planning [4]. However, integrating these functionalities into a coherent system requires reconciling the different formalisms they use to represent information and knowledge about the environment in which they operate.

Furthermore, much of the required knowledge must ultimately originate in physical sensors, but whereas deliberative functionalities tend to assume symbolic and crisp knowledge about the current state of the world, the information extracted from sensors often consists of noisy and incomplete quantitative data on a much lower level of abstraction. Thus, there is a wide gap between the information about the world normally acquired through sensing and the information that deliberative functionalities assume to be available for reasoning.

Bridging this gap is a challenging problem. It requires constructing suitable representations of the information that can be extracted from the environment using sensors and other available sources, processing the information to generate information at higher levels of abstraction, and continuously maintaining a correlation between generated representations and the environment itself. Doing this 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 standard functionalities such as various forms of image processing and information fusion as well as application-specific and possibly scenario-specific approaches. Such integration is mainly performed in an ad-hoc manner, without addressing the principles behind the integration. These principles encode parts of the current engineering knowledge of how to build this type of system.

☆ This work is partially supported by Grants from the Swedish Research Council (2005-3642, 2005-4050), the Swedish Aeronautics Research Council (NFFP4-S4203), the SSF Strategic Research Center MOVIII, the Swedish Research Council Linnaeus Center CADICS, and the Center for Industrial Information Technology CENIIT (06.09).

* Corresponding author. Tel.: +46 70 2074388.
E-mail addresses: frehe@ida.liu.se (F. Heintz), jonkv@ida.liu.se (J. Kvarnström), patdo@ida.liu.se (P. Doherty).

1474-0346/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2009.08.007

In this article, we propose using the term *knowledge processing middleware* for a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent. We claim that knowledge processing middleware should provide both a conceptual framework and an implementation infrastructure for integrating a wide variety of functionalities and managing the information that needs to flow between them. It should allow a system to incrementally process low-level sensor data and generate a coherent view of the environment at increasing levels of abstraction, eventually providing information and knowledge at a level which is natural to use in symbolic deliberative functionalities.

In addition to defining the concept of knowledge processing middleware, we describe one particular instance called *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. Streams represent aspects of the past, current, and future state of a system and its environment. Input can be provided by a wide range of distributed information sources on many levels of abstraction, while output consists of streams representing objects, attributes, relations, and events.

In the next section, a motivating example scenario is presented. Then, desirable properties of knowledge processing middleware are investigated and stream-based middleware is proposed as suitable for a wide range of systems. As a concrete example, the formal conceptual framework of our knowledge processing middleware DyKnow is described. The article is concluded with some related work and a summary.

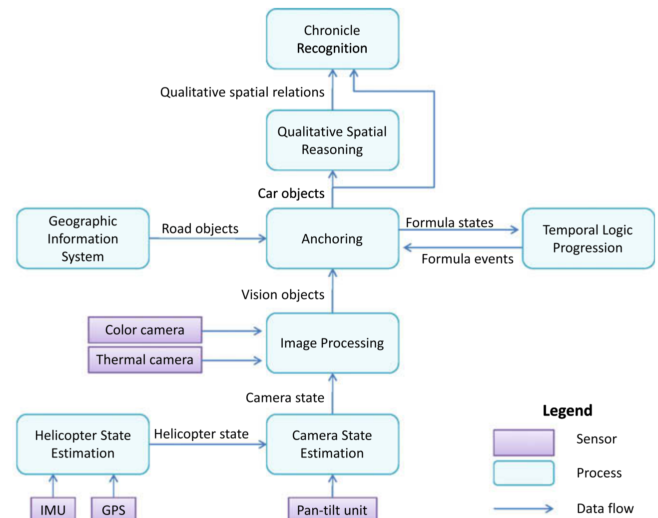


Fig. 1. Incremental processing for the traffic surveillance task.

2. A traffic monitoring scenario

Traffic monitoring is an important application domain for autonomous unmanned aerial vehicles (UAVs), providing a plethora of cases demonstrating the need for knowledge processing middleware. It includes surveillance tasks such as detecting accidents and traffic violations, finding accessible routes for emergency vehicles, and collecting traffic pattern statistics.

One possible approach to detecting traffic violations relies on describing each type of violation in a declarative formalism such as the *chronicle* formalism [1]. A chronicle defines a class of complex events as a simple temporal network [5] where nodes correspond to occurrences of events and edges correspond to metric temporal constraints between event occurrences. For example, events representing changes in high-level qualitative spatial relations such as $\mathsf{beside}(car_1, car_2)$, $\mathsf{close}(car_1, car_2)$, and $\mathsf{on}(car, road)$ might be used to detect a reckless overtake. Creating these high-level representations from low-level sensor data, such as video streams from color and thermal cameras, involves a great deal of processing at different levels of abstraction, which would benefit from being separated into distinct and systematically organized tasks.

Fig. 1 provides an overview of how the incremental processing required for the traffic surveillance task could be organized. 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 helicopter. The resulting information is fed into a *camera state estimation component*, 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 extract *vision objects* representing hypotheses regarding moving and stationary physical entities, including their approximate positions and velocities.

To use symbolic chronicle recognition, it is necessary to determine which vision objects are likely to represent cars. Such objects must be associated with car symbols in such a way that the symbol and the vision object consistently refer to the same physical object, a process known as *anchoring* [6]. This process can take advantage of knowledge about normative characteristics and behaviors of cars, such as size, speed, and the fact that cars normally travel on roads. Such characteristics can be described using formulas in a *metric temporal logic*, which are progressed (incrementally evaluated) in *states* that include current estimated car positions, velocities, and higher level predicates such as $\mathsf{on}-\mathsf{road}(car)$ and $\mathsf{in}-\mathsf{crossing}(car)$ obtained from road network information provided by a *geographic information system*. An entity satisfying the conditions can be hypothesized to be a car, a hypothesis which is subject to being withdrawn if the entity ceases to display the normative characteristics, thereby causing formula progression to signal a violation.

The next stage of processing involves deriving *qualitative spatial relations* between cars, such as $\mathsf{beside}(car_1, car_2)$ and $\mathsf{close}(car_1, car_2)$. These predicates, and the concrete events that correspond 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 scenario, which is implemented and tested on board an autonomous UAV system developed at the Unmanned Aircraft Systems Technologies (UASTech) Lab at Linköping University [7], a considerable number of distinct processes are involved in bridging the sense-reasoning gap. However, in order to fully appreciate the complexity of the system, we have to widen our perspective. Towards the smaller end of the scale, what is represented as a single process in Fig. 1 is sometimes merely an abstraction of what is in fact a set of distinct processes. Anchoring is a prime example, encapsulating a variety of tasks that could also be viewed as separate processes. 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 goal achieving procedures. Consequently, what is seen in Fig. 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 information and knowledge processing functionalities into a coherent system.

3. Design requirements

The range of functionality that could conceivably be provided by knowledge processing middleware is wide, and attempting to find a unique set of absolute requirements that could be universally agreed upon would most likely be futile. However, the following requirements have served well to guide the work presented in this article.

First, knowledge processing middleware should *permit the integration of information from distributed sources, allowing this information to be processed at many different levels of abstraction and finally transformed into a suitable form to be used in reasoning*. For traffic monitoring, sources may include cameras, barometric pressure sensors, GPS sensors, and laser range scanners as well as higher-level geographical information systems and declarative specifications of normative vehicle 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.

A second requirement is to *support both quantitative and qualitative processing*. In the traffic monitoring scenario, for example, there is a natural 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.

A third requirement is that *both bottom-up data processing and top-down model-based processing should be supported*. While each process can be dependent on "lower level" processes for its input, its output can also be used 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.

A fourth requirement is support for *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.

Support for *flexible configuration and reconfiguration* of knowledge processing is a fifth requirement. 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. For example, if a vehicle goes off-road, different forms of processing may be required.

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. Therefore a sixth requirement is for the framework to provide a *declarative specification of the information being generated and the information processing functionalities that are available*, with sufficient content to make rational trade-off decisions.

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

4. Stream-based knowledge processing middleware

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. This forms the basis of one specific type of framework called *stream-based knowledge processing middleware*, which we believe will be useful in a broad range of applications. A concrete implemented instantiation, DyKnow, will be discussed later.

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 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.

In general, streams tend to be asynchronous in nature. This can often be the case even when information is sampled and sent at regular intervals, due to irregular and unpredictable transmission delays in a distributed system. In order to minimize delays and avoid the need for frequent polling, stream implementations should be push-based and notify receiving processes as soon as new information arrives.

Using an asynchronous publish/subscribe pattern of communication decouples knowledge processes in time, space, and synchronization [8], providing a solid foundation for distributed knowledge processing applications.

For processes that do not require constant updates, such as a task planner that needs an initial state snapshot, stream generators also provide an interface for querying current and historic information generated by a process. Through integration into a unified framework, queries benefit from decoupling and asynchronicity, and lower level processing can build on a continuous stream of input before a snapshot is generated.

4.1. Streams

In an implementation, new information is added to a stream one element at a time. Formally, we instead choose to view a stream as a structure containing its own history over time. This allows us to extract a snapshot of the information received at any knowledge process at any particular point in time, which is for

example essential for the ability to validate an execution trace relative to a formal system description.

Definition 4.1 (*Stream*). A *stream* is a set of *stream elements*, where each stream element is a tuple $\langle t_a, \ldots \rangle$ whose first value, t_a, is a time-point representing the time when the element is *available* in the stream. This time-point is called the *available time* of a stream element and has to be unique within a stream. A total order < on time-points is assumed to exist.

Given a stream structure, the information that has arrived at its receiving process at a particular time-point t consists of those elements having an available time $t_a \leqslant t$. This will be used in DyKnow in Section 5.

4.2. Policies

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 should be declaratively specified. See Section 5 for concrete examples.

To satisfy the policies of the streams currently being generated, a stream generator may filter the raw output of the knowledge process and (if permitted by each policy) generate new approximated samples where necessary. Some processes may also be able to adjust their raw output (in terms of sample rate or other properties) at the request of a generator. For example, given two policies requesting samples every 200 and 300 ms, the generator might request output every 100 ms from its process. The parts of the policy that are affected by transmission through a distributed system, such as delay constraints, can also be applied by a *stream proxy* at the receiving process. This separates the generation of stream content from its adaptation.

Sometimes, it may be impossible for a stream generator to satisfy a given policy. For example, if a policy specifies a maximum transmission delay which is exceeded by the underlying communication channel, the generator can only satisfy the policy by approximating the missing value. If a subscriber sets a maximum delay and forbids approximation, it must be completely certain that the delay is never exceeded or be prepared to handle policy violations.

Definition 4.2 (*Policy*). A *policy* is a declarative specification of the desired properties of a stream, which may include advice on how to generate the stream.

4.3. Knowledge processes

A knowledge process operates on streams. Some processes take streams as input, some produce streams as output, and some do both. A process that generates stream output does so through one or more stream generators to which an arbitrary number of processes may subscribe using different policies. An abstract view of a knowledge process is shown in Fig. 2.

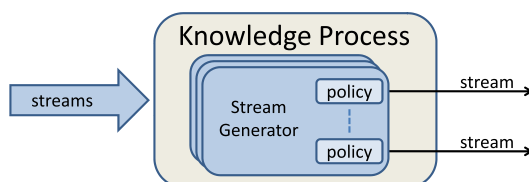


Fig. 2. A prototypical knowledge process.

Definition 4.3 (*Knowledge process*). A *knowledge process* is an active and sustained process whose inputs and outputs are in the form of streams.

Four distinct knowledge process types are identified for the purpose of modeling: Primitive processes, refinement processes, configuration processes, and mediation processes.

Primitive processes serve as interfaces to the outside world, connecting to sensors, databases, or other information sources and generating output in the form of streams. Such processes have no stream inputs but provide at least one stream generator. Sometimes the first level of data refinement may also be integrated into a primitive process. For example, image processing may be realized as a primitive process generating image streams together with a refinement process for the actual analysis, or may be integrated into a single primitive process to avoid the need for a high-bandwidth stream of live high-resolution video.

Definition 4.4 (*Primitive process*). A *primitive process* is a knowledge process that does not take any streams as input but provides output through one or more stream generators.

Refinement processes provide the main functionality of stream-based knowledge processing: The processing of streams to create more refined data, information, and knowledge. Each refinement process takes a set of streams as input and provides one or more stream generators providing stream outputs. For example, a refinement process could fuse sensor data using Kalman filters estimating positions from GPS and IMU data, or reason about qualitative spatial relations between objects.

Definition 4.5 (*Refinement process*). A *refinement process* is a knowledge process that takes one or more streams as input and provides output through one or more stream generators.

When a refinement process is created it subscribes to its input streams. For example, a position estimation process computing the position of a robot at 10 Hz could either subscribe to its inputs with the same frequency or use a higher frequency in order to filter out noise. If a middleware implementation allows a process to change the policies of its inputs during run-time, the process can dynamically tailor its subscriptions depending on the streams it is supposed to create.

In certain cases, a process must first collect information over time before it is able to compute an output. For example, a filter might require a number of measurements before it is properly initialized. This introduces a processing delay that can be remedied if the process is able to request 30 seconds of historic data, which is supported by the DyKnow implementation.

A *configuration process* provides a fine-grained form of dynamic reconfiguration by instantiating and removing knowledge processes and streams as indicated by its input.

Traffic monitoring requires position and velocity estimates for all currently monitored cars, a set that changes dynamically over time as new cars enter an area and as cars that have not been observed for some time are discarded. This is an instance of a recurring pattern where the same type of information must be produced for a dynamically changing set of objects.

This could be achieved with a static process network, where a single refinement process estimates positions for all currently visible cars. However, processes and stream policies would have to be quite complex to support more frequent updates for a specific car which is the current focus of attention.

As an alternative, a configuration process could be used to generate and maintain a dynamic network of refinement processes, where each process estimates positions for a single car. The input to the configuration process would be a single stream where each element contains the set of currently monitored cars. Whenever a new car is detected, the new (complete) set of cars is sent to the configuration process, which may create new processes. Similarly, when a car is removed, associated knowledge processes may be removed.

Definition 4.6 (*Configuration process*). A *configuration process* is a knowledge process that takes streams as inputs, has no stream generators, and creates and removes knowledge processes and streams.

Finally, a *mediation process* allows a different type of dynamic reconfiguration by aggregating or selecting information from a static or dynamic set of existing streams (leaving more complex processing of the resulting stream to refinement processes).

Aggregation is particularly useful in the fine-grained processing networks described above: If there is one position estimation process for each car, a mediation process can aggregate the outputs of these processes into a single stream to be used by those processes that do want information about all cars at once. In contrast to refinement processes, a mediation process can change its inputs over time to track the currently monitored set of cars as indicated by a stream of labels or label sets.

Selection forwards information from a particular stream in a set of potential input streams. For example, a mediation process can provide position information about the car that is the current focus of attention, automatically switching between position input streams as the focus of attention changes. Other processes interested in the current focus can then subscribe to a single semantically meaningful stream.

Definition 4.7 (*Mediation process*). A *mediation process* is a knowledge process that changes its input streams dynamically and mediates the content on the varying input streams to a fixed number of stream generators.

4.3.1. Stream generators

A knowledge process can have multiple outputs. For example, a single process may generate separate position and velocity estimates for a particular car. Each raw output is sent to a single *stream generator*, which can create an arbitrary number of output streams adapted to specific policies. For example, one process may wish to receive position estimates every 100 ms, while another may require updates only when the estimate has changed by at least 10 m.

Definition 4.8 (*Stream generator*). A *stream generator* is a part of a knowledge process that generates streams according to policies from output generated by the knowledge process.

Using stream generators separates the generic task of adapting streams to policies from the specific tasks performed by each knowledge process. Should the policies supported by a particular middleware implementation be insufficient, a refinement process can still subscribe to the unmodified output of a process and provide arbitrarily complex processing of this stream.

Note that a stream generator is not necessarily a passive filter. For example, the generator may provide information about its current output policies to the knowledge process, allowing the process to reconfigure itself depending on parameters such as the current sample rates for all output streams.

Note also that while policies are *conceptually* handled in the stream generator, an implementation may transparently move parts of policy adaptation into the stream proxy at the receiving process, thereby allowing multiple streams to be collapsed into one and reducing the associated bandwidth requirements.

5. The DyKnow framework

We will now present DyKnow, a specific stream-based middleware framework providing detailed instantiations of the generic concepts introduced in the previous sections. This provides a concrete conceptual framework for modeling knowledge processing applications (Section 5.1), with a well-defined syntax (Section 5.2) and semantics (Section 5.3). This formal framework specifies what is expected of an implementation, and can also be used by an agent to reason about its own processing.

DyKnow views the world as consisting of *objects* and *features* which may represent attributes of these objects. We specialize the general stream concept to define *fluent streams* representing an approximation of the value of a feature over time. Two concrete classes of knowledge processes are introduced: *Sources*, corresponding to primitive processes, and *computational units*, corresponding to refinement processes. A computational unit is parameterized with one or more fluent streams. Each source and computational unit provides a *fluent stream generator* which creates fluent streams from the output of the corresponding knowledge process according to *fluent stream policies*. A declarative language called KPL is used for specifying knowledge processing applications (Sections 5.2 and 5.3).

5.1. Knowledge processing domains

A knowledge processing domain defines the objects, values, and time-points used in a knowledge processing application. From them the possible fluent streams, sources, and computational units are defined. The semantics of a DyKnow knowledge processing specification is defined on an interpretation of its symbols to a knowledge processing domain.

Definition 5.1 (*Knowledge processing domain*). A *knowledge processing domain* (often referred to as a *domain*) is a tuple $\langle O, T, P \rangle$, where O is a set of objects, T is a set of time-points, and P is a set of primitive values.

Note that the temporal domain T must be associated with a total order ($<$) and operators for adding ($+$) and subtracting ($-$) time-points, and that the special value `no_value` must not occur in the O, T, or P.

Example 5.1 (*Domain*). In the remainder of this article, we will use a simplified traffic monitoring example with the domain $\langle \{o_1, \ldots, o_{10}\}, \mathbb{Z}^+, P \rangle$, where $P = \{p_1, \ldots, p_{10}\} \cup \{s_1, \ldots, s_{10}\}$ is the set of primitive values (positions and speeds) and T is the set $\mathbb{Z}^+$ of non-negative integers including zero.

5.1.1. Values, samples and fluent streams

Due to inherent limitations in sensing and processing, an agent cannot always expect access to the actual value of a feature over time but will have to use approximations. Such approximations are represented as *fluent streams*, whose elements are *samples* representing observations or estimations of the *value* of a feature at a specific point in time called the *valid time*. Each sample is also tagged with its *available time*, the time when it is ready to be processed by the receiving process after having been transmitted through a potentially distributed system.

The available time is essential when determining whether a system behaves according to specification, which depends on the information actually available as opposed to information that may have been generated but has not yet arrived. Having a specific representation of the available time also allows a process to send multiple estimates for a single valid time, for example by quickly

providing a rough estimate and then running a more time-consuming algorithm to provide a higher quality estimate. Finally, it allows us to formally model delays in the availability of a value and permits an application to use this information introspectively to determine whether to reconfigure the current processing network to achieve better performance.

Definition 5.2 (*Value*). A *simple value* in $D = \langle O, T, P\rangle$ is an object constant from O, a time-point from T, or a primitive value from P. The set of all possible simple values in D is denoted by W_D.

A *value* in D is a simple value in W_D, the special constant `no_value` indicating that a stream has no value at a particular time-point, or a tuple $\langle v_1, \ldots, v_n\rangle$ where all v_i are values, allowing the representation of structured values such as states. The set of all possible values in a domain D is denoted by V_D.

Definition 5.3 (*Sample*). A *sample* in a domain $D = \langle O, T, P\rangle$ is either the constant `no_sample` or a stream element $\langle t_a, t_v, v\rangle$, where $t_a \in T$ is its *available time*, $t_v \in T$ is its *valid time*, and $v \in V_D$ is its *value*. If $s = \langle t_a, t_v, v\rangle$ is a sample, then $atime(s) \stackrel{\text{def}}{=} t_a$, $vtime(s) \stackrel{\text{def}}{=} t_v$, and $val(s) \stackrel{\text{def}}{=} v$. The set of all possible samples in a domain D is denoted by S_D.

Example 5.2. Assume a picture p is taken by a camera source at time-point 471, and that the picture is sent through a fluent stream to an image processing process on a separate on-board computer, where it is received at time 474. This is represented as the sample $\langle 474, 471, p\rangle$.

Assume image processing extracts a set b of blobs that may correspond to vehicles. Processing finishes at time 479 and the set is sent to two distinct recipients, one on the same computer receiving it at time 482 and one on another computer receiving it at time 499. Since this information still pertains to the state of the environment at time 471, the valid time remains the same. This is represented as the two samples $\langle 482, 471, b\rangle$ and $\langle 499, 471, b\rangle$ belonging to distinct fluent streams.

The constant `no_sample` will be used to indicate that a fluent stream contains no information at a particular point in time, and can never be part of a fluent stream.

Definition 5.4 (*Fluent stream*). A *fluent stream* in a domain D is a stream where each stream element is a sample from $S_D \setminus \{\texttt{no_sample}\}$. The set of all possible fluent streams in a domain D is denoted by F_D.

Available times are guaranteed to be unique within a stream, and it is assumed that a total order < is defined on the temporal domain. Therefore, any stream $\{\langle t_{a_1}, t_{v_1}, v_1\rangle, \ldots, \langle t_{a_n}, t_{v_n}, v_n\rangle\}$ corresponds to a unique sequence $[\langle t_{a_1}, t_{v_1}, v_1\rangle, \ldots, \langle t_{a_n}, t_{v_n}, v_n\rangle]$ totally ordered by available time, and vice versa. Both notations will be used.

Example 5.3 (*Fluent stream*). Both $f_1 = \{\langle 1, 1, v_1\rangle, \langle 3, 2, v_2\rangle, \langle 4, 5, v_3\rangle\}$ and $f_2 = \{\langle 2, 1, v_4\rangle, \langle 4, 1, v_5\rangle, \langle 5, 1, v_6\rangle\}$ are valid fluent streams, visualized in Fig. 3.

For any fluent stream, we can extract the part that was available at the receiving process at any given point in time using the function $avail(f, t)$. The *last* sample that had been received at a particular time-point is given by the function $last_avail(f, t)$.

Definition 5.5. Let f be a fluent stream and t a time-point.

$$avail(f, t) \stackrel{\text{def}}{=} \{s \in f \mid atime(s) \leqslant t\}$$

$$last_avail(f, t) \stackrel{\text{def}}{=} \begin{cases} \texttt{no_sample} & \text{if } avail(f, t) = \emptyset \\ \arg\max\limits_{s \in avail(f,t)} atime(s) & \text{otherwise.} \end{cases}$$

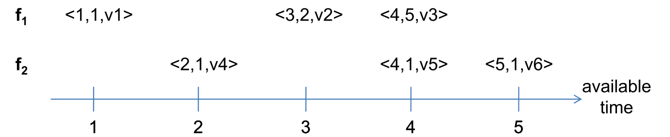


Fig. 3. Two example fluent streams.

We also need the ability to find the value of a feature at some particular *valid* time t. However, sample times may be irregular, so we cannot count on there being a sample exactly at t. Instead, $most_recent_at(f, t, t_q)$ extracts all samples available at the query time t_q with a valid time $vtime(s) \leqslant t$, selects those samples that have a maximal valid time (still $\leqslant t$), and finally (if more than one sample remains) selects the one with the greatest available time using *last_avail*. A knowledge process can then assume that this value persists until t or use filtering, interpolation, or similar techniques to find a better estimate.

Definition 5.6. Let f be a fluent stream and t, t_q time-points.

$$before(f, t, t_q) \stackrel{\text{def}}{=} \{s \in avail(f, t_q) \mid vtime(s) \leqslant t\}$$

$$most_recent_at(f, t, t_q) \stackrel{\text{def}}{=} last_avail(\{s \in before(f, t, t_q) \mid vtime(s) = \arg\max_{s' \in before(f,t,t_q)} vtime(s')\}, t_q)$$

The *prev* function returns the sample preceding a given sample in a fluent stream, or returns `no_sample` to indicate that no such sample exists. The function *availtimes* returns the set of available times for all samples in a fluent stream or a set of fluent streams.

Definition 5.7. Let f be a fluent stream and $s \in f$ be a sample. Then, $prev(f, s) \stackrel{\text{def}}{=}$

$$\begin{cases} \texttt{no_sample} & \text{if } s = \texttt{no_sample} \\ & \text{or } \neg\exists s' \in f.atime(s') < atime(s) \\ \arg\max\limits_{\substack{s' \in f \wedge \\ atime(s') < atime(s)}} atime(s') & \text{otherwise.} \end{cases}$$

Definition 5.8. Let $f, f_1, \ldots, f_n$ be fluent streams. We define:

$$availtimes(f) \stackrel{\text{def}}{=} \{atime(s) \mid s \in f\}$$

$$availtimes(\{f_1, \ldots, f_n\}) \stackrel{\text{def}}{=} \bigcup_{1 \leqslant i \leqslant n} availtimes(f_i)$$

5.1.2. Sources

A primitive process is formally modeled as a *source*, a function from time-points to samples representing the output of the process at any point in time. When the process does not produce a sample, the function returns `no_sample`.

Definition 5.9 (*Source*). Let $D = \langle O, T, P\rangle$ be a domain. A source is a function $T \mapsto S_D$ mapping time-points to samples. The set of all possible sources in D is denoted by R_D.

5.1.3. Computational units

A *computational unit* is used to model a specific type of refinement process that only considers the most recent sample from each input stream, together with its current internal state, when computing new samples for its unique output stream.

Definition 5.10 (*Computational unit*). Let $D = \langle O, T, P\rangle$ be a domain. A computational unit with arity $n > 0$, taking n inputs, is associated with a partial function $T \times S_D^n \times V_D \mapsto S_D \times V_D$ of arity $n + 2$ mapping a time-point, n input samples, and a (possibly complex) value representing the previous internal state to an

output sample and a new internal state. The set of all possible computational units for D is denoted by C_D.

The input streams to a computational unit do not necessarily contain values with synchronized valid times or available times. For example, two streams could be sampled with periods of 100 and 60 ms while a third could send samples asynchronously. In order to give the computational unit the maximum amount of information, we choose to apply its associated function whenever a new sample becomes available in any of its input streams, and to use the most recent sample in each stream. Should the unit prefer to wait for additional information, it can store samples in its internal state and return `no_sample` to indicate that no new output sample should be produced at this stage. This is defined formally using the join function.

Definition 5.11. Let $D = \langle O, T, P\rangle$ be a domain. The value of the function $join(f_1, \ldots, f_n) : F_D^n \mapsto F_D$ is the stream that is the result of joining a sequence of fluent streams $f_1, \ldots, f_n$:

$$join(f_1,\ldots,f_n) \stackrel{\text{def}}{=} \{\langle t, t, [s_1,\ldots,s_n]\rangle \mid t \in availtimes(\{f_1,\ldots,f_n\}) \wedge \quad \forall i.s_i = last_avail(f_i, t)\}.$$

Example 5.4 (*Fluent stream cont.*). Continuing Example 5.3, the result of joining the fluent streams f_1 and f_2 is the stream $[\langle 1,1,[\langle 1,1,v_1\rangle, \mathtt{no_sample}]\rangle, \langle 2,2,[\langle 1,1,v_1\rangle,\langle 2,1,v_4\rangle]\rangle, \langle 3,3,[\langle 3,2,v_2\rangle,\langle 2,1,v_4\rangle]\rangle, \langle 4,4,[\langle 4,5,v_3\rangle,\langle 4,1,v_5\rangle]\rangle, \langle 5,5,[\langle 4,5,v_3\rangle,\langle 5,1,v_6\rangle]\rangle]$, visualized in Fig. 4.

5.2. The syntax of KPL

DyKnow uses the *knowledge processing language* KPL to declaratively specify *knowledge processing applications*, static networks of primitive processes (sources) and refinement processes (computational units) connected by streams. Mediation and configuration processes modify the setup of a knowledge processing application over time and are left for future work.

The vocabulary of KPL consists of the union of two sets of symbols, the domain-independent KPL symbols and the domain-dependent symbols defined by a signature σ.

Definition 5.12 (*KPL Symbols*). The KPL symbols are comma, equals, left and right parenthesis, left and right bracket, and the KPL keywords {**any**, **approximation**, **change**, **compunit**, **delay**, **every**, **from**, **max**, **monotone**, **most**, **no**, **oo**, **order**, **recent**, **sample**, **source**, **stream**, **strict**, **strmgen**, **to**, **update**, **use**, **with**}.

Definition 5.13 (*Signature*). A *signature* σ in KPL is a tuple $\langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V}\rangle$, where

- $\mathcal{O}$ is a finite set of object symbols,
- $\mathcal{F}$ is a finite set of feature symbols with arity $\geqslant 0$,
- $\mathcal{N}$ is a finite set of stream symbols,
- $\mathcal{S}$ is a finite set of source symbols,
- $\mathcal{C}$ is a finite set of computational unit symbols, each associated with an arity > 0,
- $\mathcal{T}$ is a set of time-point symbols, and

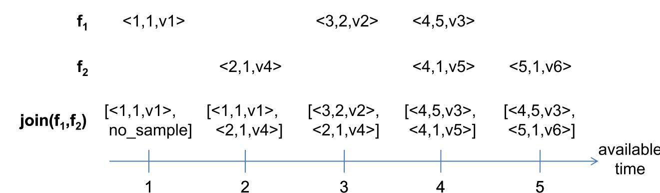


Fig. 4. An example of two fluent streams and the result of joining them.

- $\mathcal{V}$ is a finite set of value sort symbols which must include the symbols `object` and `time`.

The symbols are assumed to be unique and the sets of symbols are assumed to be disjoint.

A KPL *specification* of a knowledge processing application consists of a set of labeled statements. A source declaration, labeled **source**, declares a source corresponding to a class of primitive processes (Section 5.2.1). A computational unit declaration, labeled **compunit**, declares a parameterized computational unit corresponding to a class of refinement processes (Section 5.2.1). A fluent stream generator declaration, labeled **strmgen**, declares a specific fluent stream generator created from either a source or a computational unit (Section 5.2.2). A fluent stream declaration, labeled **stream**, declares a fluent stream generated by the application as an output (Section 5.2.3).

Definition 5.14 (*KPL Specification*). *A* KPL *specification for a signature σ is a set of source declarations, computational unit declarations, fluent stream generator declarations, and fluent stream declarations for σ.*

Example 5.5 (*KPL Example*). A small knowledge processing application estimating the speed of a single car will be used to illustrate the use of KPL. An overview is shown in Fig. 5. Formal definitions will be provided after this example.

We define the signature of this application as $\sigma = \langle\{\mathsf{car1}\}, \{\mathsf{pos/1}, \mathsf{speed/1}\}, \{\mathsf{speed_car1}\}, \{\mathsf{pos_car1}\}, \{\mathsf{SpeedEst/1}\}, \mathbb{Z}^+, \{\mathtt{pos}, \mathtt{speed}, \mathtt{time}, \mathtt{object}\}\rangle$.

The source pos_car1 provides position samples (of sort `pos`) for a car, and the fluent stream generator pos[car1] belongs to an instantiation of that source.

source `pos` pos_car1
strmgen pos[car1]=pos_car1

The computational unit SpeedEst estimates the speed of a car from a stream of positions. We create one instance of this unit, applying it to a fluent stream generated from pos[car1] using a policy that samples the source every 200 time units.

compunit `speed` SpeedEst(`pos`)
strmgen speed[car1]=SpeedEst(pos[car1] **with sample every** 200)

Finally, a named stream speed_car1, which could be used as the output of the application, is generated by the fluent stream generator speed[car1] between time-point 300 and time-point 400. **stream** speed_car1=speed[car1] **with from** 300 **to** 400.

5.2.1. Source and computational unit declarations

A *source declaration* specifies a class of primitive processes providing fluent streams of a particular value sort. A source is generally only instantiated once, since its stream generator can provide an arbitrary number of output streams.

Definition 5.15 (*Source declaration*). A *source declaration* for $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V}\rangle$ has the form **source** $v\ s$, where $v \in \mathcal{V}$ is a value sort symbol and $s \in \mathcal{S}$ is a source symbol.

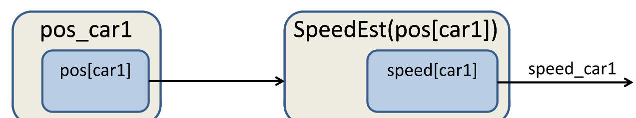


Fig. 5. Processes, stream generators, and streams in Example 5.5.

A *computational unit declaration* specifies a class of refinement processes. A computational unit can be instantiated multiple times with different input streams.

Definition 5.16 (*Computational unit declaration*). A *computational unit declaration* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form **compunit** $v_0\ c(v_1 \ldots, v_n)$, where $v_i \in \mathcal{V}$ are value sort symbols and $c \in \mathcal{C}$ is a computational unit symbol.

In the current version of KPL, each instantiation of a knowledge process is assumed to have a single unique stream generator. Therefore, the instantiation of a source or computational unit is implicit in the declaration of its generator.

5.2.2. Fluent stream generator declarations

A fluent stream generator declaration specifies both a generator and the knowledge process instance it is associated with.

The streams created by a stream generator often represent approximations of an attribute of an object or a relation between objects. The name given to a generator is therefore a structured *label* consisting of a feature symbol and zero or more object symbols, which can be used to indicate that the output of the associated process is an approximation of the value of the given feature instance over time. For example, pos[car1] is used as a label in Example 5.5, which also provides concrete examples of fluent stream generator declarations.

Definition 5.17 (*Label term*). A *label term* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form $f[o_1, \ldots, o_n]$, where $n \geqslant 0$, f is a feature symbol in $\mathcal{F}$ with arity n, and $o_1, \ldots, o_n$ are object symbols in $\mathcal{O}$. If $n = 0$, the brackets are optional.

Definition 5.18 (*Fluent stream generator declaration*). A *fluent stream generator declaration* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ is any of the following:

- **strmgen** $l = s$, where l is a label term for σ and s is a source symbol in $\mathcal{S}$.
- **strmgen** $l = c(w_1, \ldots, w_n)$, where l is a label term for σ, $n > 0$, c is a computational unit symbol in $\mathcal{C}$ with arity n, and $w_1, \ldots, w_n$ are fluent stream terms for σ.

5.2.3. Fluent stream terms and declarations

A *fluent stream term* refers to a distinct fluent stream created by a specific fluent stream generator using a specific policy.

Definition 5.19 (*Fluent stream term*). A *fluent stream term* for a signature σ has the form l **with** p, where l is a label term for σ identifying a fluent stream generator and p is a fluent stream policy specification for σ. If the policy p is the empty string, the keyword **with** can be left out.

A fluent stream term such as "pos[car1] **with sample every** 200" can be used as an input to a computational unit in a fluent stream generator declaration. In case not all processes are fully integrated into DyKnow, it can also be used to declare a named stream generated as output by an application as shown at the end of Example 5.5.

Definition 5.20 (*Fluent stream declaration*). A *fluent stream declaration* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form **stream** $n = w$, where n is a stream symbol in $\mathcal{N}$ and w is a fluent stream term for σ.

A policy specifies the desired properties of a fluent stream.

Definition 5.21 (*Fluent stream policy*). A *fluent stream policy specification* for a signature σ has the form $c_1, \ldots, c_n$, where $n \geqslant 0$ and each c_i is either an approximation constraint specification, a change constraint specification, a delay constraint specification, a duration constraint specification, or an order constraint specification for σ as defined below.

A *change constraint* specifies what must change between two consecutive samples. Given two consecutive samples, **any update** indicates that some part of the new sample must be different, while **any change** indicates that the value or valid time must be different, and **sample every** t indicates that the difference in valid time must equal the sample period t. Not specifying a change constraint is equivalent to specifying **any update**.

Definition 5.22 (*Change constraint*). A *change constraint specification* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has either the form **any update**, the form **any change**, or the form **sample every** t, where t is a time-point symbol in $\mathcal{T}$.

A *delay constraint* specifies a maximum acceptable delay, defined as the difference between the valid time and the available time of a sample. Note that delays may be intentionally introduced in order to satisfy other constraints such as ordering constraints. Not specifying a delay constraint is equivalent to specifying an infinite delay, **max delay oo**.

Definition 5.23 (*Delay constraint*). A *delay constraint specification* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form **max delay** t, where t is either the keyword **oo** or a time-point symbol in $\mathcal{T}$.

A *duration constraint* restricts the permitted valid times of samples in a fluent stream. If a duration constraint is not specified, valid times are unrestricted.

Definition 5.24 (*Duration constraint*). A *duration constraint specification* for a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ either has the form **from** t_f **to** t_t, the form **from** t_f, or the form **to** t_t, where t_f is a time-point symbol in $\mathcal{T}$ and t_t is either the keyword **oo** or a time-point symbol in $\mathcal{T}$.

An *order constraint* restricts the relation between the valid times of two consecutive samples. The constraint **any order** does not constrain valid times, while **monotone order** ensures valid times are non-decreasing and **strict order** ensures valid times are strictly increasing. Not specifying an order constraint is equivalent to specifying **any order**. A sample change constraint implies a strict order constraint.

Definition 5.25 (*Order constraint*). An *order constraint specification* for a signature σ has either the form **any order**, the form **monotone order**, or the form **strict order**.

An *approximation constraint* restricts how a fluent stream may be extended with new samples in order to satisfy its policy. If the output of a knowledge process does not contain the appropriate samples to satisfy a policy, a fluent stream generator could approximate missing samples based on available samples. The constraint **no approximation** permits no approximated samples to be added, while **use most recent** permits the addition of samples having the most recently available value as defined in Section 5.3. Not specifying an approximation constraint is equivalent to specifying **no approximation**.

In order for the stream generator to be able to determine at what valid time a sample must be produced, the **use most recent** constraint can only be used in conjunction with a complete duration constraint **from** t_f **to** t_t and a change constraint **sample every** t_s. For the stream generator to determine at what available time it should stop waiting for a sample and produce an approximation, this constraint must be used in conjunction with a delay constraint **max delay** t_d.

Definition 5.26 (*Approximation constraint*). An *approximation constraint specification* for a signature σ has either the form **no approximation** or the form **use most recent**.

5.3. The semantics of KPL

Two important entities have now been defined. A knowledge processing domain specifies the objects, time-points, and simple values available in a particular application. This indirectly defines the set of possible complex values, sources, and computational units and the fluent streams that could be produced. A KPL specification defines symbolic names for a set of sources and computational units. It also specifies how these sources and computational units are instantiated into processes, how the inputs and outputs of the processes are connected with fluent streams, and what policies are applied to these streams.

What remains is to define an *interpretation* structure for a KPL specification and to define which interpretations are *models* of the specification. However, while the KPL specification defines a specific symbol for each computational unit available for use in the application, it does not define the actual function associated with this symbol. Providing a syntactic characterization of this function in KPL would be quite unrealistic, as it would require a full description of an arbitrarily complex functionality such as image processing. We therefore assume that the interpretation of the computational unit symbols is provided in a *knowledge process specification*. In contrast, the output of a source is assumed not to be known in advance and is not specified. Thus, every possible output from a source will give rise to a (possibly empty) class of models satisfying all specifications.

Definition 5.27 (*Interpretation*). Let $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ be a signature and $D = \langle O, T, P \rangle$ be a domain. An *interpretation I* of the signature σ to the domain D is a tuple of functions $\langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$, where:

- I_O maps object symbols in $\mathcal{O}$ to distinct objects in O,
- I_F maps each feature symbol with arity n in $\mathcal{F}$ to a function $O^n \mapsto F_D$ mapping arguments to fluent streams,
- I_N maps symbols in $\mathcal{N}$ to fluent streams in F_D,
- I_S maps symbols in $\mathcal{S}$ to functions in R_D,
- I_T maps symbols in $\mathcal{T}$ to distinct time-points in T, and
- I_V maps symbols in $\mathcal{V}$ to sets of simple values in W_D.

Time-point symbols in $\mathcal{T}$ are generally assumed to be interpreted in the standard manner and associated with an addition operator +, a subtraction operator $-$, and a total order <.

Definition 5.28 (*Knowledge process specification*). Let $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ be a signature and D be a domain. Then, a *knowledge process specification* K_C for σ is a function mapping each computational unit symbol with arity n in $\mathcal{C}$ to a computational unit function in C_D with the arity $n+2$.

Example 5.6. An example interpretation of the signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ from Example 5.5 to the domain $D = \langle O, T, P \rangle$ from Example 5.1 is the tuple $\langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$, where

- I_O maps car1 to o_3.
- I_F maps pos and speed to unary functions. The function associated with pos maps o_3 (car1) to $f_1 = [\langle 250, 200, p_1 \rangle, \langle 325, 300, p_2 \rangle, \langle 430, 400, p_3 \rangle, \langle 505, 500, p_4 \rangle]$ and all other objects to the empty fluent stream. The function associated with speed maps o_3 to $f_3 = [\langle 510, 400, s_3 \rangle]$ and all other objects to the empty fluent stream.
- I_N maps speed_car1 to $f_5 = [\langle 345, 300, p_2 \rangle, \langle 460, 400, p_3 \rangle]$.
- I_S maps pos_car1 to a unary function mapping 250 to $\langle 250, 200, p_1 \rangle$, 325 to $\langle 325, 300, p_2 \rangle$, 430 to $\langle 430, 400, p_3 \rangle$, 505 to $\langle 505, 500, p_4 \rangle$, and anything else to `no_sample`.
- The standard interpretation is assumed for the temporal symbols in $\mathcal{T}$, and
- I_V maps `pos` to $\{p_1, \ldots, p_{10}\}$, `speed` to $\{s_1, \ldots, s_{10}\}$, `object` to $\{o_1, \ldots, o_{10}\}$, and `time` to $\mathbb{Z}^+$.

An example knowledge process specification for σ and D is K_C which maps SpeedEst to a computational unit taking a position p_i as input and computing the speed s_i as output.

In the following, we will assume as given a signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$, a domain $D = \langle O, T, P \rangle$, an interpretation $I = \langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$ of σ to D, a knowledge process specification K_C for σ and D, and a KPL specification s for σ.

5.3.1. Models

An interpretation satisfying all parts of a KPL specification s, given a knowledge process specification K_C, is said to be a *model* of s given K_C.

Definition 5.29 (*Model*). I is a *model* of s given K_C, written $I, K_C \models s$, iff $I, K_C \models d$ for every declaration $d \in s$.

5.3.2. Source and computational unit declarations

A source or computational unit declaration specifies the sort of the values that may be generated by a process and, for computational units, the sort of the values that may occur in its input streams. The *values* function will be used to determine which values occur in the samples of a given fluent stream.

Definition 5.30 (*Values*). Let f be a fluent stream. We define $values(f) \stackrel{\text{def}}{=} \{val(sa) \mid sa \in s \wedge sa \neq \texttt{no_sample}\}$.

Definition 5.31. Let d be a source or computational unit declaration for σ. Then, $I, K_C \models d$ according to:

$I, K_C \models$ **source** $v\ s$ iff $values(\{I_S(s)(t) \mid t \in T\}) \subseteq I_V(v)$

$I, K_C \models$ **compunit** $v_0\ c(v_1 \ldots, v_n)$ iff

$K_C(c)$ is a total function $T \times S_1 \times \ldots \times S_n \times V_D \mapsto S_0 \times V_D$

where $\{s \in S_D \mid val(s) \in I_V(v_i)\} \subseteq S_i$ for each $i \in \{1, \ldots, n\}$

and $values(S_0) \subseteq I_V(v_0)$

Example 5.7. The source declaration "**source** `pos` pos_car1" is satisfied by the interpretation I from Example 5.6, because I_S(pos_car1) is a function returning samples having values in $\{p_1, p_2, p_3, p_4\}$ which is a subset of $I_V(\texttt{pos}) = \{p_1, \ldots, p_{10}\}$.

5.3.3. Fluent stream generator declarations

Though it is conceptually the task of a fluent stream generator to adapt output streams to policies, modularity is improved by instead defining the semantics of policies in the context of each fluent stream term. The semantics of a fluent stream generator declaration will then merely ensure that the interpretation of the declared label term, which is a fluent stream, is equivalent to the unadapted output of the associated knowledge process. The following shorthand notation is used for evaluating a complete label term in an interpretation.

Definition 5.32. Let $f[o_1, \ldots, o_n]$ be a label term for σ. Then, $eval_label(I, f[o_1, \ldots, o_n]) \stackrel{\text{def}}{=} I_F(f)(I_O(o_1), \ldots, I_O(o_n))$.

A fluent stream generator declaration associates a label with the fluent stream generator for an instance of a source or a computational unit. In the case of a source, the fluent stream denoted by the label must be equivalent to the function of time denoted by the source symbol, which can be defined as follows:

Definition 5.33. Let s be a source symbol for σ and l a label term for σ. Then, $I, K_C \models$ **strmgen** $l = s$ iff $eval_label(I, l) = \{sa \mid \exists t . I_S(s)(t) = sa \wedge sa \neq \texttt{no_sample}\}$.

A computational unit calculates an output sample whenever there is a new input sample in either of its input streams. This is equivalent to calculating an output sample for each tuple of samples in the *join* of its input streams. For the purpose of modeling, each sample calculation requires as input the current time, the sequence of input samples, and the previous internal state, generating as output a tuple containing a new sample and the new internal state. As explained in the following section, the *eval_fsterm* function returns a set of possible fluent streams corresponding to a fluent stream term.

Definition 5.34. Let l be a label term for σ, c be a computational unit symbol for σ, $fsterm_1, \ldots, fsterm_m$ be fluent stream terms, and $i_0 = \texttt{no_value}$ be the initial internal state for c. For brevity, we introduce the notation $\bar{s}$ to denote a sequence of samples of appropriate length for its context. Then,

$$I, K_C \models \textbf{strmgen } l = c(fsterm_1, \ldots, fsterm_m)$$
$$\text{iff } eval_label(I, l) = \{s_1, \ldots, s_n\} \text{ where there exists}$$
$$f_1 \in eval_fsterm(I, fsterm_1), \ldots, f_m \in eval_fsterm(I, fsterm_m)$$
$$\text{such that } join(f_1, \ldots, f_m) = [\langle t_1, t_1, \bar{s}_1\rangle, \ldots, \langle t_n, t_n, \bar{s}_n\rangle]$$
$$\text{and for each } j \in \{1, \ldots, n\}, \langle s_j, i_j\rangle = K_C(c)(t_j, \bar{s}_j, i_{j-1})$$

Example 5.8. The declaration "**strmgen** pos[car1]=pos_car1" is satisfied by the interpretation I from Example 5.6, because $I_F(\text{pos})(I_O(\text{car1})) = f_1$ which contains the same samples as the function denoted by pos_car1.

5.3.4. Fluent stream terms and declarations

A fluent stream term consists of a label term l and a (possibly empty) policy p, and refers to a stream created by one particular subscription to a fluent stream generator. Such a stream is generated from the output of a knowledge process by actively adapting it to a policy, and in certain cases, this can be done in more than one way. A fluent stream term is therefore evaluated to a *set* of possible streams denoted by $eval_fsterm(I, l \textbf{ with } p)$, giving implementations some freedom in choosing how policies are applied while still ensuring that all constraints are met.

The interpretation of the fluent stream term is defined in two steps. First, there are cases where new samples must be added to the stream in order to approximate missing values. The function $extend(f, p)$ is introduced for this purpose. Intuitively, it computes the valid times when the fluent stream must have values in order to satisfy the policy and approximates any missing values. Second, a maximal set of samples which satisfies the policy p must be filtered out from the stream, leaving only the final adapted stream.

Definition 5.35. Let l **with** p be a fluent stream term. The interpretation of the fluent stream term, $eval_fsterm(I, l \textbf{ with } p)$ is then defined as follows:

$$eval_fsterm(I, l \textbf{ with } p) \stackrel{\text{def}}{=}$$
$$\{f \in satisfying(I, l \textbf{ with } p) \mid \neg\exists f' \in satisfying(I, l \textbf{ with } p). f \subset f'\}$$
$$satisfying(I, l \textbf{ with } p) \stackrel{\text{def}}{=}$$
$$\{f \sqsubseteq extend(eval_label(I, l), p) \mid I, f \models p\}$$
$$extend(f, p) \stackrel{\text{def}}{=}$$
$$\begin{cases} \{\langle t_a, t_v, v\rangle \mid & \text{if } \textbf{usemostrecent} \in p \\ \exists n \geqslant 0. t_v = b + sn \wedge t_v \leqslant e & \wedge \exists s. \textbf{sample every } s \in p \\ \wedge t_a = t_v + d & \wedge \exists b, e. \textbf{from } b \textbf{ to } e \in p \\ \wedge v = most_recent_at(f, t_v, t_a)\} & \wedge \exists d. \max \textbf{ delay } d \in p \\ f & \text{otherwise.} \end{cases}$$

The substream relation $\sqsubseteq$ used above is similar but not identical to the subset relation. Informally, a fluent stream f' is a substream of f if f' could be created from f by removing some samples and delaying others. Thus, for every sample $\langle t'_a, t_v, v\rangle$ in the "new" stream f', there must be a sample $\langle t_a, t_v, v\rangle \in f$ with $t_a \leqslant t'_a$ having the same value and valid time.

Definition 5.36 (*Substream relation).* A fluent stream f' is a substream of a fluent stream f, written $f' \sqsubseteq f$, iff $\forall \langle t'_a, t_v, v\rangle \in f' \; \exists t_a. \; [t_a \leqslant t'_a \wedge \langle t_a, t_v, v\rangle \in f]$.

Example 5.9 (*Substream relation*). In Example 5.6, the stream $f_5 = [\langle 345, 300, p_2\rangle, \langle 460, 400, p_3\rangle]$ is a substream of $f_1 = [\langle 250, 200, p_1\rangle, \langle 325, 300, p_2\rangle, \langle 430, 400, p_3\rangle, \langle 505, 500, p_4\rangle]$, since each sample in f_5 has a corresponding sample in f_1 where only the available time differs and is earlier in f_1.

The entailment relation $I, f \models p$, used in the *satisfying* function above, determines which streams satisfy a particular policy.

Definition 5.37. Let f be a fluent stream and p a fluent stream policy specification for σ. Then, $I, f \models p$ according to:

$$I, f \models c_1, \ldots, c_n \text{ iff } I, f \models c_1 \text{ and } \ldots \text{ and } I, f \models c_n$$
$$I, f \models \textbf{no approximation} \text{ iff } \texttt{true}$$
$$I, f \models \textbf{use most recent} \text{ iff } \texttt{true} \quad (\text{handled by } extend)$$
$$I, f \models \textbf{any update} \text{ iff } \texttt{true}$$
$$I, f \models \textbf{any change} \text{ iff } \forall s, s' \in f \; [s' = prev(f, s) \rightarrow vtime(s) \neq vtime(s') \vee val(s) \neq val(s')]$$
$$I, f \models \textbf{sample every } t \text{ iff } \forall s, s' \in f \; [s' = prev(f, s) \rightarrow vtime(s) - vtime(s') = I_T(t)]$$
$$I, f \models \textbf{from } t_f \textbf{ to } t_t \text{ iff } \forall s \in f \; [I_T(t_f) \leqslant vtime(s) \leqslant I_T(t_t)]$$
$$I, f \models \textbf{from } t_f \textbf{ to oo} \text{ iff } \forall s \in f \; [I_T(t_f) \leqslant vtime(s)]$$
$$I, f \models \max \textbf{ delay } t \text{ iff } \forall s \in f \; [atime(s) - vtime(s) \leqslant I_T(t)]$$
$$I, f \models \textbf{any order} \text{ iff } \texttt{true}$$
$$I, f \models \textbf{monotone order} \text{ iff } \forall s, s' \in f \; [s' = prev(f, s) \rightarrow vtime(s') \leqslant vtime(s)]$$
$$I, f \models \textbf{strict order} \text{ iff } \forall s, s' \in f \; [s' = prev(f, s) \rightarrow vtime(s') < vtime(s)]$$

Example 5.10 (*Fluent stream policy*). The fluent stream policy specification "**sample every** 100" is satisfied by the fluent stream f_1 from Example 5.6, since the difference between each pair of valid times is exactly 100 time units. The same stream does not satisfy the policy "**max delay** 40" since the first sample has a delay of 50 time units.

Example 5.11 (*Fluent stream term*). Let $f = [\langle 280, 200, p_1\rangle, \langle 480, 400, p_3\rangle]$ be a fluent stream. Using the interpretation I from Example 5.6, f is one of the possible streams that can be generated from the fluent stream term "pos[car1] **with sample every** 200". First, f is a substream of the stream f_1 denoted by the label l as shown in Example 5.9. Second, f satisfies p since the difference between each pair of valid times is exactly 200 time units. Finally, it is not possible to add further samples to f without violating p.

Finally, we define the semantics of a stream declaration.

Definition 5.38. Let n be a stream symbol for σ, l a label term for σ, and p a fluent stream policy specification. Then, $I, K_C \models \textbf{stream } n = l \textbf{ with } p$ iff $I_N(n) = eval_fsterm(I, l \textbf{ with } p)$.

5.4. Extensions to DyKnow

DyKnow is an extensible knowledge processing middleware framework where additional functionalities can be incorporated

if they are judged sufficiently important for a range of applications. The following extensions should be mentioned: A *state synchronization* mechanism generates temporally synchronized states from separate unsynchronized fluent streams [9]. A *formula progression* component incrementally evaluates metric temporal logic formulas over states [10]. An *object classification and anchoring* system builds *object linkage structures* where objects are incrementally classified and re-classified as new information becomes available [11]. A *chronicle recognition system* detects complex high-level spatio-temporal events given a declarative description [7]. Finally, a group of agents can share information by setting up a *DyKnow federation* which allows a knowledge processing application to import and export streams [12]. These functionalities are implemented and can be used by any knowledge processing application.

6. The DyKnow implementation

To support distributed real-time and embedded systems, the formal DyKnow framework specified in the previous section has been implemented as a CORBA middleware service [13]. The CORBA event notification service [14] is used to implement streams and to decouple knowledge processes from clients subscribing to their output. This provides an implementation infrastructure for knowledge processing applications.

The DyKnow service takes a set of KPL declarations and sets up the required processing and communication infrastructure to allow knowledge processes to work according to specification in a distributed system. It can take a complete KPL application specification resulting in an application where the set of knowledge processes and the set of streams connecting them are static. It is also possible to specify an initial KPL specification and then incrementally add new declarations. The implementation has been extensively used as part of an autonomy architecture on board two Yamaha RMAX UAVs in the UASTech Lab at Linköping University [15]. For details see [9].

7. Related work

There are many frameworks and toolkits for supporting the development of robotic systems. These often focus on how to support the integration of different functional modules into complete robotic systems. To handle this integration, most approaches support distributed computing and communication. However, even when a framework supports communication between distributed components it does not necessarily explicitly support information and knowledge processing.

There are a few surveys available [16–19]. Of these the survey by Kramer and Scheutz [17] is the most detailed. It evaluates nine freely available robotic development environments according to their support for specification, platforms, infrastructure, and implementation. However, it mainly focuses on software engineering aspects while we are more interested in how robotic frameworks support knowledge processing and bridging the sense-reasoning gap.

Before summarizing the support for knowledge processing in current robotics software frameworks we will give an overview of a representative selection of such frameworks.

7.1. CAST/BALT

The CoSy architecture schema toolkit (CAST) is developed in order to study different instantiations of the CoSy architecture schema [20,21]. An architecture schema defines a design space containing many different *architectural instantiations* which are specific architecture designs. CAST implements the instantiations of the architecture schema using the Boxes and Lines Toolkit (BALT) which provides a layer of component connection software.

BALT provides a set of processes which can be connected by either 1-to-1 *pull* connections or 1-to-*N* *push* connections. With its support for push connections, distributing information, and integrating reasoning components BALT can be seen as basic stream-based knowledge processing middleware. A difference is that it does not provide any declarative specification to control push connections nor does it explicitly represent time.

An instantiation of the CoSy architecture schema (CAS) consists of a collection of interconnected subarchitectures (SAs). Each subarchitecture contains a set of processing components that can be connected to sensors and actuators, and a working memory which acts like a blackboard within the subarchitecture. A processing component can either be *managed* or *unmanaged*. An unmanaged processing component runs constantly and directly pushes its results onto the working memory. A managed process, on the other hand, monitors the working memory content for changes and suggests new 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.

When data is written to a working memory, *change objects* are propagated to all subarchitecture-managed components and all connected working memories, which forward the objects to the managed components in their subarchitectures. This is the primary mechanism for distributing information through the architecture. Like many other frameworks, CAST has no explicit support for time. Support for specifying the information a component is interested in is also limited to the change objects.

7.2. GenoM

GenoM was developed for use in distributed hybrid robotic architectures, and provides a language for specifying functional modules and automatically generating module implementations according to a generic model [22,23]. A *module* is a software entity offering *services* related to a physical (sensor, actuator) or a logical (data) *resource*. The module is responsible for its resource, including detecting and recovering from failures.

Services are parameterized and activated asynchronously through *requests* to the module. A request starts a client/server relationship that ends with the termination of the service by the server returning a reply to the client. The reply includes an execution report from a set of predefined reports and optionally data produced by the service. A service may dynamically choose to use other services to execute a request.

During the execution of a service it may have to read or produce data. This is done using *posters*. A poster is a structured shared memory that is readable by any other element in the architecture but only writable by its owner. Each poster always provides the most up to date value which can be accessed by a service using the unique poster identifier. This allows services to access the value of any poster in their own pace. Since a service can be executed periodically it is possible to poll a poster with a certain sample period.

From a knowledge processing perspective, GenoM provides support for both synchronous and asynchronous polling of data but not for asynchronous notification of the availability of new data. Neither does it provide support for synchronization or the specification of the data required by a service.

7.3. MARIE

The mobile and autonomous robotics integration environment (MARIE) is a middleware framework for developing and integrating

new and existing software for robotic systems [24,25]. MARIE uses a mediator interoperability layer (MIL) adapted from the mediator design pattern to implement distributed applications using heterogeneous software. The MIL acts as a centralized control unit which interacts with any number of *components* to coordinate the global interactions among them. Components have *ports* that can be connected to other components. Each component has a director port used to control execution, a configuration port, and possibly input and output ports. The configuration port is used to configure the input and output ports.

To integrate a set of applications into a working system a component called an *adapter* must be developed for each application. Adapters can then be connected either using the existing communication mechanisms supported by MARIE or through communication adapters such as mailboxes, splitters, shared maps, and switches. A mailbox serves as a buffer between asynchronous components. A splitter forwards incoming data to multiple components. A shared map is a push-in/pull-out key-value data structure used to store data that can be used by other components at their own rate. A switch takes multiple inputs but only sends one of them to its single output port.

To our knowledge MARIE provides no support for a component to specify the desired properties of its input which DyKnow supports through its policies. Neither does it have any explicit representation of time or synchronization of input streams. It might, however, be possible to provide synchronization through the use of a communication adapter.

7.4. Discussion

Even though there exist many different frameworks on different abstraction level, the support provided for knowledge processing is usually limited to providing distributed components that can communicate. Most frameworks provide some form of publish/subscribe communication while some also support the combination of push and pull communication. What these frameworks lack is a way for a consumer to specify the information that it would like to have. For example, in DyKnow a knowledge process can specify the start and end time of a stream or the sampling period and how to approximate missing values. To our knowledge, there is no robotics framework that supports this type of specification. Some of the frameworks do however provide ways of specifying when a value has changed enough for it to be relevant for a consumer, for example CAST.

Another important aspect of knowledge processing that is not supported by the mentioned frameworks is an explicit representation of time. In DyKnow all samples are tagged with both valid time and available time. This makes it possible for a knowledge process to reason about when a value is valid, when it was actually available to the process, and how much it was delayed by previous processing. DyKnow also supports sample-based sources that periodically read a sensor or a database and make the result available through a stream generator. Stream generators support caching of historic data which can be used for later processing. This allows the specification of streams that begin in the past, where the content is partially generated from cached historic data.

Since DyKnow explicitly tags samples with time-stamps and each stream is associated with a declarative policy specifying its properties, it is possible to define a synchronization algorithm that extracts a sequence of states from a set of asynchronous streams. Each of these states is valid at a particular time-point and contains a single value from each of the asynchronous streams valid at the same time as the state. Some of these values may be approximated as specified by the state synchronization policy. This is another example of functionality that is missing from existing approaches.

8. Discussion

In the introduction six requirements on knowledge processing middleware were presented (Section 3). These requirements are not binary in the sense that a system either satisfies them or not. Instead, most systems satisfy the requirements to some degree. In this section, we argue that DyKnow provides a significant degree of support for each of the six requirements.

8.1. Support integration of information from distributed sources

DyKnow satisfies this requirement by virtue of three features: a CORBA-based implementation, explicit support for representing time, and a stream synchronization mechanism that uses the declarative policies to determine how to synchronize a set of asynchronous streams and derive a stream of states.

8.2. Support processing on many levels of abstraction

General support is provided through fluent 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 also 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. Heintz and Doherty [26] argue that this support is enough to provide an appropriate framework for supporting all the functional abstraction levels in the JDL Data Fusion Model.

8.3. 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 [10] and chronicle recognition [7] can be integrated using DyKnow.

8.4. Support quantitative and qualitative processing

Fluent 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.

8.5. 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.

8.6. 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 fluent stream. This information can then be used by knowledge processes subscribing to this fluent 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.

8.7. 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.

Mediation and configuration processes provide ample support for flexible reconfiguration where streams and processes are added and removed at run time. The DyKnow implementation also provides the necessary interfaces to procedurally reconfigure an application.

8.8. 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.

9. 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, called stream-based knowledge processing middleware, 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, without unnecessarily limiting the class of systems to which it is applicable. Finally, DyKnow was presented to give an example of a concrete instantiation.

Knowledge processing middleware such as DyKnow is an engineering tool for developing applications that need to derive information and knowledge from sensor data and use it to reason about the embedding environment. The tool allows existing engineering knowledge in the form of both knowledge representation and software components to be integrated in a knowledge processing application using a formally defined communication and integration framework.

References

[1] M. Ghallab, On Chronicles: representation, on-line recognition and learning, in: Proceedings of KR, 1996, pp. 597–607.
[2] M. Wzorek, P. Doherty, Reconfigurable path planning for an autonomous unmanned aerial vehicle, in: Proceedings of ICAPS, 2006.
[3] P.O. Pettersson, P. Doherty, Probabilistic roadmap based path planning for an autonomous unmanned helicopter, Journal of Intelligent and Fuzzy Systems 17 (5) (2006).
[4] J. Kvarnström, P. Doherty, TALplanner: a temporal logic based forward chaining planner, Annals of Mathematics and Artificial Intelligence 30 (2000) 119–169.
[5] R. Dechter, I. Meiri, J. Pearl, Temporal constraint networks, Artificial Intelligence 49 (1991) 61–95.
[6] S. Coradeschi, A. Saffiotti, An introduction to the anchoring problem, Robotics and Autonomous Systems 43 (2-3) (2003) 85–96.
[7] F. Heintz, P. Rudol, P. Doherty, From images to traffic behavior – a UAV tracking and monitoring operation, in: Proceedings of Fusion, 2007.
[8] P.T. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computer Survey 35 (2) (2003) 114–131.
[9] F. Heintz, DyKnow: A Stream-Based Knowledge Processing Middleware Framework, Ph.D. Thesis, Linköpings Universitet, 2009. http://www.ida.liu.se/frehe/publications/thesis.pdf.
[10] P. Doherty, J. Kvarnström, F. Heintz, A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems, Journal of Autonomous Agents and Multi-Agent Systems (forthcoming).
[11] F. Heintz, J. Kvarnström, P. Doherty, A stream-based hierarchical anchoring framework, in: Proceedings of IROS, 2009.
[12] F. Heintz, P. Doherty, DyKnow federations: distributing and merging information among UAVs, in: Proceedings of Fusion, 2008.
[13] Object Management Group, The CORBA Specification v 3.1, 2008.
[14] P. Gore, D.C. Schmidt, C. Gill, I. Pyarali, The design and performance of a real-time notification service, in: Proceedings of the 10th IEEE Real-time Technology and Application Symposium, 2004.
[15] P. Doherty, P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, B. Wingman, A distributed architecture for autonomous unmanned aerial vehicle experimentation, in: Proceedings of DARS, 2004.
[16] A. Orebäck, H. Christensen, Evaluation of architectures for mobile robotics, Autonomous Robots 14 (1) (2003) 33–49.
[17] J. Kramer, M. Scheutz, Development environments for autonomous mobile robots: a survey, Autonomous Robots 22 (2) (2007) 101–132.
[18] G. Biggs, B. Macdonald, A survey of robot programming systems, in: Proc. of the Australasian Conference on Robotics and Automation, 2003.
[19] N. Mohamed, J. Al-Jaroodi, I. Jawhar, Middleware for robotics: a survey, in: Proceedings of RAM, 2008.
[20] N. Hawes, M. Zillich, J. Wyatt, BALT & CAST: middleware for cognitive robotics, in: Proceedings of RO-MAN, 998–1003, 2007a.
[21] N. Hawes, A. Sloman, J. Wyatt, M. Zillich, H. Jacobsson, G.-J. Kruijff, M. Brenner, G. Berginc, D. Skočaj, Towards an integrated robot with multiple cognitive functions, in: Proceedings of AAAI, 2007b.
[22] A. Mallet, S. Fleury, H. Bruyninckx, A specification of generic robotics software components: future evolutions of GenoM in the orocos context, in: Proceedings of IROS, 2002, pp. 2292–2297.
[23] S. Fleury, M. Herrb, R. Chatila, GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture, in: Proceedings of IROS, 1997, pp. 842–848.
[24] C. Côté, Y. Brosseau, D. Létourneau, C. Ra, F. Michaud, Robotic software integration using MARIE, International Journal of Advanced Robotic Systems 3 (1) (2006) 55–60.
[25] C. Côté, D. Létourneau, F. Michaud, Robotics system integration frameworks: MARIE's approach to software development and integration, in: Springer Tracts in Advanced Robotics: Software Engineering for Experimental Robotics, vol. 30, Springer Verlag, 2007.
[26] F. Heintz, P. Doherty, DyKnow: a knowledge processing middleware framework and its relation to the JDL Data Fusion Model, Journal of Intelligent and Fuzzy Systems 17 (4) (2006) 335–351.