Stream Reasoning in DyKnow: A Knowledge Processing Middleware System*

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

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

Abstract. The information available to modern autonomous systems is often in the form of streams. As the number of sensors and other stream sources increases there is a growing need for incremental reasoning about the incomplete content of sets of streams in order to draw relevant conclusions and react to new situations as quickly as possible. To act rationally, autonomous agents often depend on high level reasoning components that require crisp, symbolic knowledge about the environment. Extensive processing at many levels of abstraction is required to generate such knowledge from noisy, incomplete and quantitative sensor data. We define *knowledge processing middleware* as a systematic approach to integrating and organizing such processing, and argue that connecting processing components with *streams* provides essential support for steady and timely flows of information. DyKnow is a concrete and implemented instantiation of such middleware, providing support for stream reasoning at several levels. First, the formal KPL language allows the specification of streams connecting knowledge processes and the required properties of such streams. Second, chronicle recognition incrementally detects complex events from streams of more primitive events. Third, complex metric temporal formulas can be incrementally evaluated over streams of states. DyKnow and the stream reasoning techniques are described and motivated in the context of a UAV traffic monitoring application.

1 Introduction

Modern autonomous systems usually have many sensors producing continuous streams of data. As the systems become more advanced the number of sensors grow, as exemplified by the humanoid robot CB^2 which has 2 cameras, 2 microphones, and 197 tactile sensors [1]. Further communication streams are produced when such systems interact. Some systems may also be connected to the Internet and have the opportunity to access streams of online information such as weather reports, news about the area, and so on. The fact that much of this information is available in the form of streams highlights the growing need for advanced stream processing capabilities in autonomous systems, where one can incrementally reason about the incomplete content of a set of streams in order to draw new conclusions as quickly as possible. This is in contrast to many of

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

the current techniques used in formal knowledge representation and reasoning, which assume a more or less static knowledge base of facts to be reasoned about.

Additionally, 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. We use 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 believe that a stream-based approach to knowledge processing middleware is appropriate. To demonstrate the feasibility we have developed DyKnow, a fully implemented stream-based framework providing both conceptual and practical support for structuring a knowledge processing system as a set of streams and computations on streams [2]. The properties of each stream is specified by a declarative *policy*. 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. DyKnow also explicitly supports two techniques for incremental reasoning with streams: Chronicle recognition for detecting complex events and progression of metric temporal logic to incrementally evaluate temporal logical formulas.

DyKnow and the stream reasoning techniques are described and motivated in the context of a UAV traffic monitoring application.

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 stream reasoning and 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.

Suppose a human operator is trying to maintain situational awareness about traffic in an area using static and mobile sensors such as surveillance cameras together with an unmanned helicopter. Reducing the amount of information sent to the operator also reduces her cognitive load, helping her to focus her attention on salient events. Therefore, each sensor platform should monitor traffic situations and only report back relevant high-level events, such as reckless overtakes and probable drunk driving.

Traffic violations, or other events to be detected, should be represented formally and declaratively. This can be done using *chronicle recognition* [3], where each chronicle defines a parameterized class of complex events as a simple temporal network [4] whose

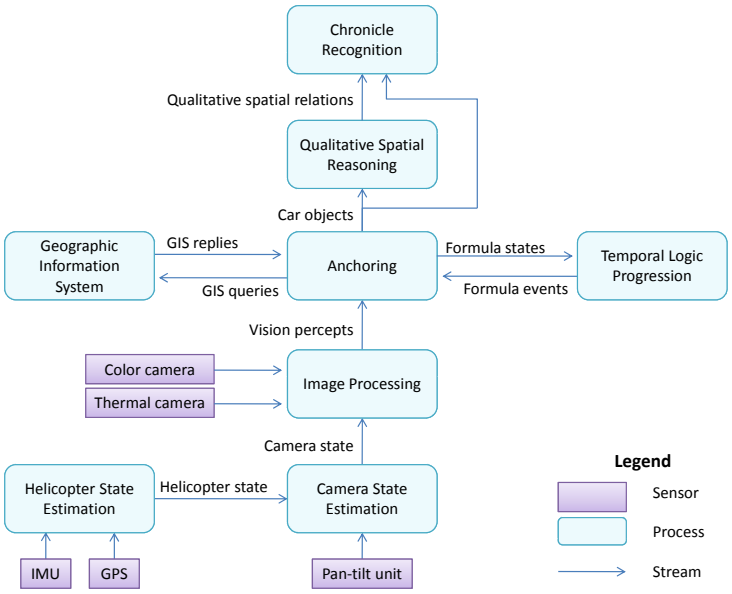

Fig. 1. An overview of how the processing required for traffic surveillance could be organized.

nodes correspond to occurrences of high-level qualitative events and edges correspond to metric temporal constraints. For example, events representing changes in qualitative spatial relations such as beside(car_1, car_2), close(car_1, car_2), and on($car_1, road_7$) 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 extensive information and knowledge processing within each sensor platform.

Fig. 1 provides an overview of how part of the incremental processing required for the traffic surveillance task could be organized as a set of distinct DyKnow knowledge processes. At the lowest level, a *helicopter state estimation component* uses data from an *inertial measurement unit* (IMU) and a *global positioning system* (GPS) to generate a stream of position and attitude estimates. A *camera state estimation component* uses this information, together with a stream of states from the *pan-tilt unit* on which the cameras are mounted, to generate a stream of current camera states. The *image processing component* uses the camera state stream to determine where the camera is currently pointing. Video streams from the *color* and *thermal cameras* can then be analyzed in order to generate a stream of *vision percepts* representing hypotheses about moving and stationary physical entities, including their approximate positions and velocities.

Symbolic formalisms such as chronicle recognition require a consistent assignment of symbols, or identities, to the physical objects being reasoned about and the sensor data received about those objects. This is a process known as *anchoring* [5]. Image analysis may provide a partial solution, with vision percepts having symbolic identities that persist over short intervals of time. However, changing visual conditions or objects temporarily being out of view lead to problems that image analysis cannot (and should not) handle. This is the task of the anchoring component, which uses *progression* over

a stream of states to evaluate potential hypotheses expressed as formulas in a metric temporal logic. The anchoring system also assists in object classification and in the extraction of higher level attributes of an object. For example, a *geographic information system* can be used to determine whether an object is currently on a road or in a crossing. Such attributes can in turn be used to derive streams of relations *between* objects, including *qualitative spatial relations* such as $\mathsf{beside}(car_1, car_2)$ and $\mathsf{close}(car_1, car_2)$. Streams of concrete events corresponding to changes in these predicates and attributes finally provide sufficient information for the *chronicle recognition system* to determine when higher-level events such as reckless overtakes occur.

3 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 for *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 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 [6], providing a solid foundation for distributed knowledge processing applications.

For processes that do not require constant updates, such as an automated task planner that needs an initial state snapshot, stream generators also provide a query interface to retrieve current and historic information generated by a process. Integrating such queries into the same framework allows them to benefit from decoupling and asynchronicity and permits lower level processing to build on a continuous stream of input before a snapshot is generated.

3.1 Streams

Intuitively, a stream serves as a communication channel between two knowledge processes, where elements are incrementally added by a source process and eventually arrive at a destination process. Verifying whether the contents such a stream satisfies a specific policy requires a formal model. For simplicity, we define a stream as a snapshot containing its own history up to a certain point in time, allowing us to determine exactly which elements had arrive at any preceding time. This is essential for the ability to validate an execution trace relative to a formal system description.

Definition 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.*

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 $t_a \leq t$.

3.2 Policies

Each stream is associated with a *policy* specifying a set of requirements on its contents. Such requirements may include the fact 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. A policy can also give advice on how to ensure that these requirements are satisfied, for example by indicating how to handle missing or excessively delayed values. For introspection purposes, policies should be declaratively specified. Concrete examples are given in Section 4.

Each subscription to a stream generator includes a specific policy to be used for the generated stream. The stream generator can use this policy to filter the output of a knowledge process or forward it to the process itself to control its internal setup. Those parts of the policy that are affected by transmission through a distributed system, such as constraints on delays, can also be used by a *stream proxy* at the receiving process. This separates the generation of stream content from its adaptation.

Definition 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.*

3.3 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.

Definition 3 (Knowledge process). *A knowledge process is an active and sustained process whose inputs and outputs are in the form of streams.*

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

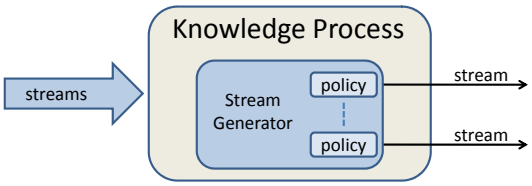

Fig. 2. A prototypical knowledge process

Primitive Processes *Primitive processes* serve as interfaces to the outside world, connecting to sensors, databases, or other information sources and providing their output in the form of streams. Such processes have no stream inputs but provide a non-empty set of stream generators. Their complexity may range from simple adaptation of external data into a stream-based framework to more complex tasks such as image processing.

Definition 4 (Primitive process). *A primitive process is a knowledge process without input streams that provides output through one or more stream generators.*

Refinement Processes The main functionality of stream-based knowledge processing middleware is to process streams to create more refined data, information, and knowledge. This type of processing is done by a *refinement process* which takes a set of streams as input and provides one or more stream generators providing stream outputs. For example, a refinement process could do image processing, fuse sensor data using Kalman filters estimating positions from GPS and IMU data, or reason about qualitative spatial relations between objects.

Definition 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.

Configuration Processes 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, one can use a dynamic network of processes, where each refinement process estimates positions for a single car. A *configuration process* provides a fine-grained form of dynamic reconfiguration by instantiating and removing knowledge processes and streams as indicated by its input.

Definition 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.*

For traffic monitoring, 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.

Mediation Processes 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.

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 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.*

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 meters.

Definition 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 generic 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.

4 DyKnow

DyKnow is a concrete instantiation of the generic stream-based middleware framework defined in the previous section. DyKnow provides both a conceptual framework for modeling knowledge processing and an implementation infrastructure for knowledge processing applications. The formal framework can be seen as a specification of what is expected of the implementation infrastructure. It can also be used by an agent to reason about its own processing. A detailed formal description of DyKnow is available in [2].

DyKnow views the world as consisting of *objects* and *features*, where features may for example represent attributes of objects. The general stream concept is specialized 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* creating fluent streams from the output of the corresponding knowledge process according to *fluent stream policies*. The declarative language KPL is used for specifying knowledge processing applications.

DyKnow is implemented as a CORBA middleware service. It uses the CORBA event notification service [7] to implement streams and to decouple knowledge processes from clients subscribing to their output. See [2] for the details.

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 knowledge processing specification is defined on an interpretation of its symbols to a knowledge processing domain.

Definition 9 (Knowledge processing domain). *A knowledge processing domain is a tuple $\langle O, T, V \rangle$, where O is a set of objects, T is a set of time-points, and V is a set of values.*

4.1 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*, a specialization of the previously introduced stream structure where elements are *samples*. Each sample represents an observation or estimation of the value of a feature at a specific point in time called the *valid time*. Like any stream element, a 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 10 (Sample). *A sample in a domain $D = \langle O, T, V \rangle$ 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 $v \in V$ is its value. The set of all possible samples in a domain D is denoted by S_D.*

Example 1. 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 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 of blobs is sent to two distinct recipients, one receiving it at time 482 and one receiving it at time 499. This information still pertains to the state of the environment at time 471, and therefore 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 11 (Fluent stream). *A fluent stream in a domain D is a stream where each stream element is a sample from $S_D \setminus \{$*`no_sample`*$\}$.*

4.2 Sources

Primitive processes can be used to provide interfaces to external data producers or sensors, such as the GPS, IMU, and cameras on a UAV. A primitive process is formally modeled as a *source*, a function from time-points to samples representing the output of the primitive process at any point in time. If the function returns `no_sample`, the primitive process does not produce a sample at the given time.

Definition 12 (Source). *Let $D = \langle O, T, V \rangle$ be a domain. A source is a function $T \mapsto S_D$ mapping time-points to samples.*

4.3 Computational Units

Refinement processes are used to perform computations on streams, ranging from simple addition of integer values to Kalman filters, image processing systems, and even more complex functions. In a wide variety of cases, only a single output stream is required (though this stream may consist of complex values). It is also usually sufficient to have access to the current internal state of the process together with the most recent sample of each input stream to generate a new output sample. A process of this type can be modeled as a *computational unit*.

Definition 13 (Computational unit). *Let $D = \langle O, T, V \rangle$ be a domain. A computational unit with arity $n > 0$, taking n inputs, is associated with a partial function $T \times S_D^n \times V \mapsto S_D \times V$ of arity $n + 2$ mapping a time-point, n input samples, and a value representing the previous internal state to an output sample and a new internal state.*

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 ms 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.

4.4 Fluent Stream Policies

A policy specifies the desired properties of a fluent stream and is defined as a set of constraints on the fluent stream. There are five types of constraints: Approximation, change, delay, duration, and order constraints.

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*.

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.

A *duration constraint* restricts the allowed valid times of samples in a fluent stream.

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. A sample change constraint implies a strict order constraint.

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.

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.

4.5 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. For details of KPL including the formal semantics see [2].

Definition 14 (KPL Grammar).

```
KPL_SPEC ::= ( SOURCE_DECL | COMP_UNIT_DECL
             | FSTREAM_GEN_DECL | FSTREAM_DECL )+
SOURCE_DECL ::= source SORT_SYM SOURCE_SYM
COMP_UNIT_DECL ::= compunit SORT_SYM
                   COMP_UNIT_SYM '(' SORT_SYM ( ',' SORT_SYM )* ')'
FSTREAM_GEN_DECL ::= strmgen LABEL '='
                     ( SOURCE_SYM
                     | COMP_UNIT_SYM '(' FSTREAM_TERM ( ',' FSTREAM_TERM )* ')' )
FSTREAM_DECL ::= stream STREAM_SYM '=' FSTREAM_TERM
LABEL ::= FEATURE_SYM ( '[' OBJECT_SYM ( ',' OBJECT_SYM )* ']' )?
FSTREAM_TERM ::= LABEL ( with FSTREAM_POLICY )?
FSTREAM_POLICY ::= STREAM_CONSTR ( ',' STREAM_CONSTR )*
STREAM_CONSTR ::= APPRX_CONSTR | CHANGE_CONSTR | DELAY_CONSTR
                | DURATION_CONSTR | ORDER_CONSTR
APPRX_CONSTR ::= no approximation | use most recent
CHANGE_CONSTR ::= any update | any change | sample every TIME_SYM
DELAY_CONSTR ::= max delay ( TIME_SYM | oo )
DURATION_CONSTR ::= from TIME_SYM | ( from TIME_SYM )? to ( TIME_SYM | oo )
ORDER_CONSTR ::= any order | monotone order | strict order
```

5 Chronicle Recognition

Many applications of autonomous vehicles involve surveillance and monitoring where it is crucial to recognize *events* related to objects in the environment. For example, a UAV monitoring traffic must be able to recognize events such as a car overtaking another, a car stopping at an intersection, and a car parking next to a certain building.

We can classify events as being either *primitive* or *complex*. A primitive event is either directly observed or grounded in changes in feature values, while a complex event is defined as a spatio-temporal pattern of other events. The purpose of an event

recognition system is to detect complex events from a set of observed or previously detected events. In the traffic monitoring domain, for example, the complex event of car A *overtaking* car B can be defined in terms of a chain of events where a car A is first behind, then left of, and finally in front of car B together with temporal constraints on the events such as the total overtake should take less than 120 seconds.

One formalism for expressing complex events is the chronicle formalism which represents and detects complex events described in terms of temporally constrained events [3]. The chronicle recognition algorithm takes a stream of time-stamped event occurrences and finds all matching chronicle instances as soon as possible. This makes it a good example of a stream reasoning technique. To do this, the algorithm keeps track of all possible developments in an efficient manner by compiling chronicles into simple temporal constraint networks [4]. To detect chronicle instances, the algorithm keeps track of all partially instantiated chronicle models. To begin with each chronicle model is associated with a completely uninstantiated instance. Each time a new event is received it is checked against all the partial instances to see if it matches any previously unmatched event. If that is the case, then a copy of the instance is created and the new event is integrated into the temporal constraint network by instantiating the appropriate variables and propagating all constraints [3]. This propagation can be done in polynomial time since the temporal constraint network is simple. It is necessary to keep the original partial chronicle instance to match a chronicle model against all subsets of event occurrences. If all the events have been matched then a complete instance has been found. Recognized instances of a chronicle can be used as events in another chronicle. The chronicle recognition algorithm is complete as long as the observed event stream is complete, i.e. any change of a value of an attribute is captured by an event.

Time is considered a linearly ordered discrete set of instants, whose resolution is sufficient to represent the changes in the environment. Time is represented by time-points and all the interval constraints permitted by the restricted interval algebra [8] are allowed. This means that it is possible to represent relations such as before, after, equal, and metric distances between time-points but not their disjunctions.

We have used chronicle recognition in the traffic monitoring application to detect traffic patterns [9].

6 Progression of Metric Temporal Logic

First order logic is a powerful technique for expressing complex relationships between objects. Metric temporal logics extends first order logics with temporal operators that allows metric temporal relationships to be expressed. For example, our temporal logic, which is a fragment of the Temporal Action Logic (TAL) [10], supports expressions which state that a formula F should hold within 30 seconds and that a formula F' should hold in every state between 10 and 20 seconds from now. This fragment is similar to the well known Metric Temporal Logic [11]. Informally, $\Diamond_{[\tau_1,\tau_2]}\,\phi$ ("eventually") holds at τ iff ϕ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$, while $\Box_{[\tau_1,\tau_2]}\,\phi$ ("always") holds at τ iff ϕ holds at all $\tau' \in [\tau+\tau_1, \tau+\tau_2]$. Finally, $\phi\,\mathsf{U}_{[\tau_1,\tau_2]}\,\psi$ ("until") holds at τ iff ψ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$ such that ϕ holds in all states in (τ, τ').

The semantics of these formulas are defined over infinite state sequences. To make metric temporal logic suitable for stream reasoning, the formulas are incrementally

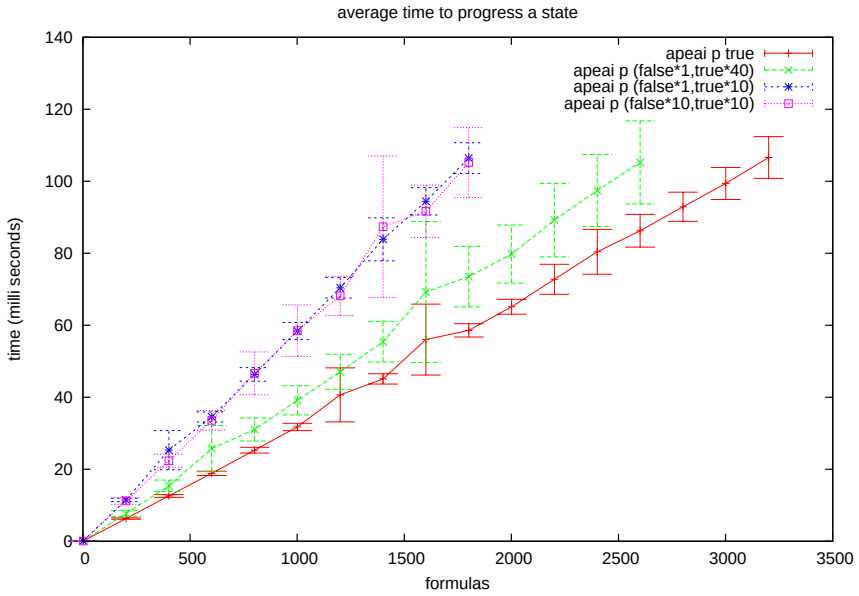

Fig. 3. Testing: Always Not p Implies Eventually Always p (average progression time).

evaluated by DyKnow using progression over a timed state stream. The result of progressing a formula through the first state in a stream is a new formula that holds in the remainder of the state stream iff the original formula holds in the complete state stream. If progression returns true (false), the entire formula must be true (false), regardless of future states. See Heintz [2] for formal details.

DyKnow also provides support for generating streams of states synchronizing distributed streams. Using their associated policies it is possible to determine when the best possible state at each time-point can be extracted.

Even though the size of a progressed formula may grow exponentially in the worst case, many common formulas do not. One example is the formula $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} p$, corresponding to the fact that if p is false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true. To estimate the cost of evaluating this formula, it was progressed through several different state streams corresponding to the best case, the worst case, and two intermediate cases. A new state in the stream was generated every 100 ms, which means that all formulas must be progressed within this time limit or the progression will fall behind. The results in Fig. 3 shows that 100 ms is sufficient for the progression of between 1500 and 3000 formulas of this form on the computer on-board our UAV, depending on the state stream.

We have used this expressive metric temporal logic to monitor the execution of complex plans in a logistics domain [12] and to express conditions for when to hypothesize the existence and classification of observed objects in an anchoring module [2]. For example in execution monitoring, suppose that a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to

τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification: $\Box\, \forall uav.(\mathsf{power}(uav) > M \rightarrow \mathsf{power} < f \cdot M\, \mathsf{U}_{[0,\tau]}\, \Box_{[0,\tau']}\, \mathsf{power}(uav) \leq M)$

7 Related Work

The conceptual stream reasoning architecture proposed by Della Valle et al [13] consists of four stages: Select, Abstract, Reason, and Decide. The Select component uses filtering and sampling to select a subset of the available streams. These streams are then processed by the Abstract component to turn data into richer information by converting the content to RDF streams. The Reason component takes these RDF streams and reasons about their content. Finally, the Decide component evaluates the output and determines if the result is good enough or if some of the previous stages have to be adapted and further data processed.

Compared to this framework, DyKnow provides support for all four stages to a varying degree without restricting itself to serial processing of the four steps. The policies and the computational units provide tools for selection and abstraction, with particular support from the anchoring module to associated symbols to sensor data. The chronicle recognition component and the formula progression engine are two particular stream reasoning techniques that can be applied to streams.

In general, stream reasoning is related to many areas, since the use of streams is common and have many different uses. Some of the most closely related areas are data stream management systems [14–16], publish/subscribe middleware [6, 17], event-based systems [18–21], complex event processing [22, 23], and event stream processing [24]. Even though most of these systems provide some contributions to stream reasoning few of them provide explicit support for lifting the abstraction level and doing general reasoning on the streams. The approaches that come the closest are complex event processing, but they are limited to events and do not reason about objects or situations.

8 Conclusions

We have presented DyKnow, a stream-based knowledge processing middleware framework, and shown how it can be used for stream reasoning. Knowledge processing middleware is a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent. Since knowledge processing is fundamentally incremental in nature it is modeled as a set of active and sustained knowledge processes connected by streams where each stream is specified by a declarative policy.

DyKnow is a concrete and implemented instantiation of such middleware, providing support for stream reasoning at several levels. First, the formal KPL language allows the specification of streams connecting knowledge processes and the required properties of such streams. Second, chronicle recognition incrementally detects complex events from streams of more primitive events. Third, complex metric temporal formulas can be incrementally evaluated over streams of states using progression.

Since DyKnow is a general framework providing both conceptual and implementation support for stream processing it is easy to add new functionality and further improve its already extensive support for stream reasoning.

References

1. Minato, T., Yoshikawa, Y., Noda, T., Ikemoto, S., Ishiguro, H., Asada, M.: Cb2: A child robot with biomimetic body for cognitive developmental robotics. In: Proceedings of IEEE/RAS International Conference on Humanoid Robotics. (2007)
2. Heintz, F.: DyKnow: A Stream-Based Knowledge Processing Middleware Framework. PhD thesis, Linköpings universitet (2009)
3. Ghallab, M.: On chronicles: Representation, on-line recognition and learning. In: Proceedings of KR. (1996) 597–607
4. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. AIJ **49** (1991)
5. Coradeschi, S., Saffiotti, A.: An introduction to the anchoring problem. Robotics and Autonomous Systems **43**(2-3) (2003) 85–96
6. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2) (2003) 114–131
7. Gore, P., Schmidt, D.C., Gill, C., Pyarali, I.: The design and performance of a real-time notification service. In: Proc. of Real-time Technology and Application Symposium. (2004)
8. Nebel, B., Burckert, H.J.: Reasoning about temporal relations: A maximumal tractable subclass of allen's interval algebra. Journal of ACM **42**(1) (1995) 43–66
9. Heintz, F., Rudol, P., Doherty, P.: From images to traffic behavior – a UAV tracking and monitoring application. In: Proceedings of Fusion'07. (2007)
10. Doherty, P., Kvarnström, J.: Temporal action logics. In Lifschitz, V., van Harmelen, F., Porter, F., eds.: The Handbook of Knowledge Representation. Elsevier (2007)
11. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4) (1990) 255–299
12. Doherty, P., Kvarnström, J., Heintz, F.: A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Journal of Autonomous Agents and Multi-Agent Systems (2009)
13. Valle, E.D., Ceri, S., Barbieri, D., Braga, D., Campi, A.: A First step towards Stream Reasoning. In: Proceedings of the Future Internet Symposium. (2008)
14. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. VLDB Journal (August 2003)
15. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of PODS'02. (2002)
16. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: Proceedings of CIDR'03. (2003)
17. OMG: The data-distribution service specification v 1.2 (jan 2007)
18. Carzaniga, A., Rosenblum, D.R., Wolf, A.L.: Challenges for distributed event services: Scalability vs. expressiveness. In: Engineering Distributed Objects. (1999)
19. Pietzuch, P.: Hermes: A Scalable Event-Based Middleware. PhD thesis, University of Cambridge (2004)
20. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems **19**(3) (2001) 332–383
21. Cugola, G., Nitto, E.D., Fuggetta, A.: The jedi event-based infrastructure and its application to the development of the opss wfms. IEEE Trans. Softw. Eng. **27**(9) (2001) 827–850
22. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston, MA, USA (2002)
23. Gyllstrom, D., Wu, E., Chae, H.J., Diao, Y., Stahlberg, P., Anderson, G.: Sase: Complex event processing over streams. In: Proceedings of CIDR'07. (2007)
24. Demers, A., Gehrke, J., Biswanath, P., Riedewald, M., Sharma, V., White, W.: Cayuga: A general purpose event monitoring system. In: Proceedings of CIDR'07. (2007)