Linköping Studies in Science and Technology

Dissertation No. 1240

DyKnow: A Stream-Based Knowledge Processing Middleware Framework

by

Fredrik Heintz

Linköping University
INSTITUTE OF TECHNOLOGY

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

Linköping 2009

ISBN 978–91–7393–696–5
ISSN 0345–7524
Printed by LiU-Tryck, Linköping, Sweden

Dedicated to the loving memory of
Anneli Dahlström (1975-2005)

Abstract

As robotic systems become more and more advanced the need to integrate existing deliberative functionalities such as chronicle recognition, motion planning, task planning, and execution monitoring increases. To integrate such functionalities into a coherent system it is necessary to reconcile the different formalisms used by the functionalities to represent information and knowledge about the world. To construct and integrate these representations and maintain a correlation between them and the environment it is necessary to extract information and knowledge from data collected by sensors. However, deliberative functionalities tend to assume symbolic and crisp knowledge about the current state of the world while the information extracted from sensors often is noisy and incomplete quantitative data on a much lower level of abstraction. There is a wide gap between the information about the world normally acquired through sensing and the information that is assumed to be available for reasoning about the world.

As physical autonomous systems grow in scope and complexity, bridging the gap in an ad-hoc manner becomes impractical and inefficient. Instead a principled and systematic approach to closing the sense-reasoning gap is needed. At the same time, a systematic solution has to be sufficiently flexible to accommodate a wide range of components with highly varying demands. We therefore introduce the concept of *knowledge processing middleware* for a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent. A set of requirements that all such middleware should satisfy is also described.

A stream-based knowledge processing middleware framework called DyKnow is then presented. Due to the need for incremental refinement of information at different levels of abstraction, computations and processes within the stream-based knowledge processing framework are modeled as active and sustained *knowledge processes* working on and producing *streams*. DyKnow supports the generation of partial and context dependent stream-based representations of past, current, and potential future states at many levels of abstraction in a timely manner.

To show the versatility and utility of DyKnow two symbolic reasoning engines are integrated into DyKnow. The first reasoning engine is a metric temporal logical progression engine. Its integration is made possible by extending DyKnow with a state generation mechanism to generate state sequences over which temporal logical formulas can be progressed. The second reasoning engine is a chronicle

recognition engine for recognizing complex events such as traffic situations. The integration is facilitated by extending DyKnow with support for anchoring symbolic object identifiers to sensor data in order to collect information about physical objects using the available sensors. By integrating these reasoning engines into DyKnow, they can be used by any knowledge processing application. Each integration therefore extends the capability of DyKnow and increases its applicability.

To show that DyKnow also has a potential for multi-agent knowledge processing, an extension is presented which allows agents to federate parts of their local DyKnow instances to share information and knowledge.

Finally, it is shown how DyKnow provides support for the functionalities on the different levels in the JDL Data Fusion Model, which is the de facto standard functional model for fusion applications. The focus is not on individual fusion techniques, but rather on an infrastructure that permits the use of many different fusion techniques in a unified framework.

The main conclusion of this thesis is that the DyKnow knowledge processing middleware framework provides appropriate support for bridging the sense-reasoning gap in a physical agent. This conclusion is drawn from the fact that DyKnow has successfully been used to integrate different reasoning engines into complex unmanned aerial vehicle (UAV) applications and that it satisfies all the stated requirements for knowledge processing middleware to a significant degree.

Acknowledgment

Writing a PhD thesis is about maturing as a researcher. I was naive and full of myself when I started. I thought I had everything it takes to make a quick and glorious career. I had breezed through my Master's program, I was ambitious, and I was ready to work hard. I learned the hard way that research is not a crusade against ignorant and narrow minded researchers who intentionally misunderstand your work, but rather a matter of presenting, motivating, and marketing your ideas and solutions to make smart and well informed researchers understand and accept you. It is my job to convince them that what I am saying is interesting and important. I hope this thesis is a step in that direction.

Today, I am still ambitious and ready to work hard, but I have realized that good research is as much about communication as about solving problems. Finding and proving the perfect solution to a difficult problem will not be a breakthrough until the scientific community has understood and accepted the importance of the problem and the quality of the solution. Each iteration of this thesis has improved the scientific quality of the content, but more importantly the accessibility of the ideas and the results. They have transformed the thesis from a Joycean stream of consciousness to the text it is today.

I am immensely grateful to my supervisor Patrick Doherty not only for providing a highly stimulating and rewarding research environment, but most importantly for forcing me to make myself clear. One of the most profound insights I have gained while writing this thesis is how unaware I was of my own communication. I did not really reflect over how I said things and how the shaping of a message affects the result of the message. Thank you Patrick.

I am forever thankful to Jonas Kvarnström for his patient, thorough, detailed, and constructive criticism on all parts of the thesis. Jonas, you deserve all the credit you can get. Without your help this thesis might not have been at all, and if it had been it would not have been nearly as good as it is now. Thank you Jonas.

Björn Wingman and Tommy Persson have played an important role as research engineers in the work with this thesis. Björn implemented the progression engine used in the thesis and both Tommy and Björn have been very helpful regarding all aspects of implementing the software used in the thesis.

I would like to thank Patrik Haslum, David Landén, Martin Magnusson, Per Nyblom, Per-Magnus Olsson and Tommy Persson for fruitful (and sometimes frustrating) discussions and for reading and commenting on drafts of the thesis. I would

also like to thank everyone who has been involved in the WITAS project and in the hard but rewarding work on our different UAV platforms, especially Gianpaolo Conte, Torsten Merz, Per Olof Pettersson, Piotr Rudol, and Mariusz Wzorek. Finally I would like to thank Jenny Ljung for making work more fun and everyone else at AIICS and IDA.

Last, but not least, I thank my parents Lennart and Christina, my grandmother Ingrid, my sister Maria, my brother Anders, my girlfriend Anne, my friends Michael, Johan, Jim, Sissel, Mikael, Mattias, Fredrik and Victor, and the rest of my family for sometimes pushing me and for sometimes not asking about my progress, but most importantly for your unconditional love and support.

Thank you all!

This thesis is dedicated to the ever loving memory of Anneli Dahlström (1975-2005) who tragically passed away while doing what she loved. Our paths were intertwined ever since my first visit to HG in Nolle-P 1996 when she enchanted me with her energy, charm, and wits. She was my strongest supporter in times of despair and always knew how to live life to the fullest. It makes me sad that she is not around to see this thesis finished. She would have been the happiest and proudest person of us all. She was a part of me and when she died that piece of me was lost forever. However, the piece of her that is a part of me will keep on living.

The work in this thesis has been generously supported by the Wallenberg laboratory for research on Information Technology and Autonomous Systems (WITAS) funded by the Wallenberg Foundation, the Swedish Aeronautics Research Council (NFFP), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR) grant 2005-3642, and the Swedish Research Council Linnaeus Center CADICS.

List of Publications

The contributions in this thesis are based on the following publications.

Heintz, F. 2001. Chronicle recognition in the WITAS UAV project – A preliminary report. In *SAIS 2001, Working notes*.

Heintz, F., and Doherty, P. 2004. DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems* 15(1):3–13.

Heintz, F., and Doherty, P. 2004. Managing dynamic object structures using hypothesis generation and validation. In *Proceedings of the AAAI Workshop on Anchoring Symbols to Sensor Data*.

Doherty, P.; Haslum, P.; **Heintz, F.**; Merz, T.; Nyblom, P.; Persson, T.; and Wingman, B. 2004. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 221–230.

Heintz, F., and Doherty, P. 2005. DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In Dunin-Keplicz, B.; Jankowski, A.; Skowron, A.; and Szczuka, M., eds., *Monitoring, Security, and Rescue Techniques in Multiagent Systems*, Advances in Soft Computing, 479–492. Springer Verlag.

Heintz, F., and Doherty, P. 2005. A knowledge processing middleware framework and its relation to the JDL data fusion model. In Blasch, E., ed., *Proceedings of the Eighth International Conference on Information Fusion (Fusion'05)*.

Heintz, F., and Doherty, P. 2005. A knowledge processing middleware framework and its relation to the JDL data fusion model. In Ögren, P., ed., *Proceedings of the Swedish Workshop on Autonomous Robotics (SWAR'05)*.

Heintz, F., and Doherty, P. 2006. DyKnow: A knowledge processing middleware framework and its relation to the JDL data fusion model. *Journal of Intelligent and Fuzzy Systems* 17(4):335–351.

Heintz, F.; Rudol, P.; and Doherty, P. 2007. From images to traffic behavior – A UAV tracking and monitoring application. In *Proceedings of the 10th International Conference on Information Fusion (Fusion'07)*.

Heintz, F.; Rudol, P.; and Doherty, P. 2007. Bridging the sense-reasoning gap using DyKnow: A knowledge processing middleware framework. In Hertzberg, J.; Beetz, M.; and Englert, R., eds., *KI 2007: Advances in Artificial Intelligence*, volume 4667 of *LNAI*, 460–463. Springer Verlag.

Heintz, F., and Doherty, P. 2008. DyKnow federations: Distributing and merging information among UAVs. In *Proceedings of the 11th International Conference on Information Fusion (Fusion'08)*.

Kvarnström, J.; **Heintz, F.**; and Doherty, P. 2008. A temporal logic-based planning and execution monitoring system. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*.

Heintz, F.; Kvarnström, J.; and Doherty, P. 2008. Bridging the sense-reasoning gap: DyKnow – A middleware component for knowledge processing. In *Proceedings of the IROS workshop on Current software frameworks in cognitive robotics integrating different computational paradigms*.

Heintz, F.; Kvarnström, J.; and Doherty, P. 2008. Knowledge processing middleware. In Carpin, S.; Noda, I.; Pagello, E.; Reggiani, M.; and von Stryk, O., eds., *Proceedings of the first international conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, volume 5325 of *LNAI*, 147–158. Springer Verlag.

Doherty, P.; Kvarnström, J.; and **Heintz, F.** 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Automated Agents and Multi-Agent Systems* Forthcoming. Springer Verlag.

Contents

Part I

Introduction and Background

Chapter 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 functionalities for reasoning about the world have already been developed, including chronicle recognition, motion planning, task planning, and execution monitoring. However, in order to integrate these functionalities into a coherent system it is necessary to reconcile the different formalisms they use to represent information and knowledge about the world and the environment in which they are supposed to operate.

Furthermore, much of the required information and 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 about the world.

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 primitive sources, processing the information to generate information at higher levels of abstraction, and continuously maintaining a correlation between generated information 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 more or less standard functionalities such as various forms of image processing and information fusion as well as application-specific and possibly even scenario-specific approaches. Such integration is today mainly performed in an ad hoc manner, without addressing the principles behind the integration.

In this thesis, 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. It is called knowledge processing because the result of processing could be interpreted as knowledge by an 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.

Besides defining the concept of knowledge processing middleware, this thesis describes one particular instance called *DyKnow*. DyKnow is a stream-based knowledge processing middleware framework providing software support for creating streams representing 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. By using DyKnow to develop knowledge processing systems, conceptual and practical support is provided for structuring these as a set of streams and computations on streams. The output of such a system is a set of streams representing objects, attributes, relations, and events.

The research in this thesis is part of a larger effort to build intelligent autonomous unmanned aerial vehicles (UAVs) capable of carrying out complex missions. The research began as part of the Wallenberg Laboratory for Information Technology and Autonomous Systems (WITAS), a very successful basic research initiative. The main objective of WITAS was the development and integration of hardware and software for a vertical take-off and landing platform for fully autonomous missions (Doherty et al., 2000; Doherty, 2004). An experimental autonomous UAV platform was developed based on the Yamaha RMAX helicopter and it has been used to demonstrate several fully autonomous capabilities. The platform has been tested in applications such as traffic monitoring and surveillance, emergency services assistance, and photogrammetry and surveying.

When the project associated with WITAS was completed a new research lab, the Unmanned Aircraft Systems Technologies (UASTech) Lab, was formed to continue the research. Our UAV platform is therefore referred to as the UASTech UAV platform. A picture of the platform is shown in Figure 1.1. Some of the implemented functionalities are autonomous take off and landing (Merz, Duranti, and Conte, 2004), trajectory following in three dimensions (Conte, 2007), generation of collision free trajectories by a probabilistic path planner (Pettersson, 2006; Wzorek and Doherty, 2006), generation of plans to achieve complex goals using a task planner (Kvarnström, 2005), online monitoring of the execution of plans (Doherty, Kvarnström, and Heintz, 2009), finding human bodies using image processing (Rudol and Doherty, 2008), tracking cars using image processing (Heintz, Rudol, and Doherty, 2007b), and recognizing complex events using a chronicle recognition system (Heintz, Rudol, and Doherty, 2007b). Several of these functionalities are described in this thesis.

The research methodology used within our group is scenario-based, where very

Figure 1.1: The UASTech UAV platform based on the Yamaha RMAX helicopter.

challenging scenarios out of reach of current systems are specified and serve as long term goals to drive both theoretical and applied research. Most importantly, attempts are always made to close the theory/application loop by implementing and integrating results in our UAVs and deploying them for empirical testing at an early stage. We then iterate and continually increase the robustness and functionality of the components.

We therefore start by introducing two challenging example scenarios. The scenarios demonstrate the need and use of knowledge processing middleware since they both require the integration of different sensing and reasoning functionalities in order to achieve their mission goals. The first example is a traffic monitoring scenario which uses a chronicle recognition functionality to detect complex traffic patterns (Heintz, Rudol, and Doherty, 2007b). The second example is an emergency service scenario which uses planning and execution monitoring functionalities to deliver supplies to injured people in a disaster situation (Doherty and Rudol, 2007; Doherty, Kvarnström, and Heintz, 2009). Both scenarios have been implemented to a large extent using our UAV platform and will be described in more detail later in the thesis.

1.1 Motivating Scenarios

Unmanned aerial vehicles are becoming commonplace in both civil and military applications, especially for missions which are considered dull, dirty, or dangerous. One important application domain for UAVs is surveillance. An example of a surveillance mission is flying over unknown and potentially hostile areas to build terrain models, which might be dangerous. Another example is to quickly get

an overview of a disaster area which might be dirty due to chemical or nuclear contamination. This mission could also include helping rescue services find injured people and deliver medical supplies. A third example is to help law enforcement agencies to monitor some area or some people for ongoing or potential criminal activity. This is often a dull activity, which may cause human pilots or operators to lose their attention and focus. Therefore it would be beneficial if it could be done by autonomous UAVs.

To complete these complex missions a UAV must continuously gather information from many different sources. Examples of sources are sensors, databases, other UAVs, and human operators. The UAV must select relevant information for the ongoing tasks and derive higher-level knowledge about the environment and the UAV itself to correctly interpret what is happening and to make appropriate decisions. In other words, the UAV must create and maintain its own situational awareness and do it in time for the results to be useful. Achieving situational awareness usually requires information from many sources on different abstraction levels to be processed and integrated in order to get an accurate understanding of the environment. This is a task that knowledge processing middleware is designed to support.

1.1.1 A Traffic Monitoring Scenario

Traffic monitoring is an important application domain for research in autonomous unmanned aerial vehicles, which provides 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 statistics about traffic patterns.

Suppose a human operator is trying to maintain situational awareness about traffic in an area using static and mobile sensors such as surveillance cameras and our Yamaha RMAX. One approach to solving this problem would be for the sensor platforms to relay videos and other data to the operator for human inspection. Another, more scalable, approach would be for each sensor platform to monitor traffic situations which arise and only report back relevant high-level events, such as reckless overtakes and drunk driving. Only reporting high-level events would reduce the amount of information sent to the operator and thereby reduce the cognitive load on the operator. This would help the operator to focus her attention on salient events. At the same time, recognizing high-level events would require more information and knowledge processing within each sensor platform. This type of processing can be facilitated by knowledge processing middleware, such as DyKnow.

In the case of detecting traffic violations, one possible approach relies on using a formal declarative description of each type of violation. A violation can for example be represented using a chronicle (Ghallab, 1996). A chronicle defines a class of complex events as a simple temporal network (Dechter, Meiri, and Pearl, 1991) where nodes correspond to occurrences of high-level qualitative events and edges correspond to metric temporal constraints between event occurrences. For

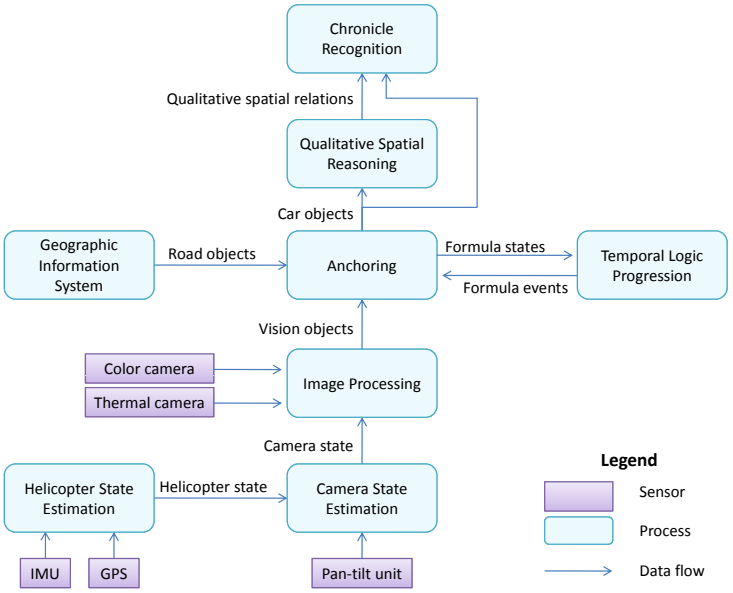


Figure 1.2: An overview of how the incremental processing required for the traffic surveillance task could be organized.

example, to detect a reckless overtake, 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. 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.

Figure 1.2 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 the symbolic chronicle formalism, each individual car has to be represented with a symbol. An important problem is therefore to associate vision objects with car symbols in such a way that both the symbol and the vision object refer to the same physical object, a process known as *anchoring* (Coradeschi and Saffiotti,

2003).

It is therefore necessary to further reason about the type and identity of each vision object. This could for example be done using knowledge about normative characteristics of cars, such as size, speed, and driving behaviors. One interesting approach to describing such characteristics relies on the use of formulas in a *metric temporal modal logic*, which are incrementally *progressed* through *states* that include current estimated car positions, velocities, and other relevant information. 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 the formula progression component to signal a violation.

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

In this example, we can identify a considerable number of distinct processes involved in bridging the gap between sensing and reasoning and generating the necessary symbolic representations from sensor data. However, in order to fully appreciate the complexity of the system, we have to widen our perspective somewhat. Looking towards the smaller end of the scale, we can see that what is represented as a single process in Figure 1.2 is sometimes merely an abstraction of what is in fact a set of distinct 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 Figure 1.2 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.

As argued in the remainder of the introduction, 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. Starting in Chapter 3 we introduce a general approach to knowledge processing middleware based on streams. In Chapter 4 DyKnow, a concrete stream-based knowledge processing middleware framework, is presented. How DyKnow can be realized and implemented as a CORBA middleware service is described in Chapter 5. The UAV platform is presented in Chapter 6 and progression of metric temporal logical formulas in Chapter 7. Finally, in Chapter 8 it is shown how the full scenario can be implemented

by bringing all the different components together.

1.1.2 An Emergency Service Scenario

On December 26, 2004, a devastating earthquake of high magnitude occurred off the west coast of Sumatra. This resulted in a tsunami which hit the coasts of India, Sri Lanka, Thailand, Indonesia, and many other islands. Both the earthquake and the tsunami caused great devastation. During the initial stages of the catastrophe, there was a great deal of confusion and chaos in setting into motion rescue operations in such wide geographic areas. The problem was exacerbated by a shortage of manpower, supplies, and machinery. The highest priorities in the initial stages of the disaster were searching for survivors in many isolated areas where road systems had become inaccessible and providing relief in the form of delivery of food, water, and medical supplies.

Let us assume that one has access to a fleet of autonomous unmanned helicopter systems with ground operation facilities. How could such a resource be used in the real-life scenario described?

A prerequisite for the successful operation of this fleet would be the existence of a multi-agent (UAV platforms, ground operators, etc.) software infrastructure for assisting emergency services in such a catastrophe situation. At the very least, one would require the system to allow mixed initiative interaction with multiple platforms and ground operators in a robust, safe, and dependable manner. As far as the individual platforms are concerned, one would require a number of different capabilities, not necessarily shared by each individual platform, but by the fleet in total. These capabilities would include:

- the ability to scan and search for salient entities such as injured humans, building structures, or vehicles;
- the ability to monitor or surveil these salient points of interest and continually collect and communicate information back to ground operators and other platforms to keep them situationally aware of current conditions; and
- the ability to deliver supplies or resources to these salient points of interest if required. For example, identified injured persons should immediately receive a relief package containing food, water, and medical supplies.

To be more specific in terms of the scenario, we can assume there are two separate legs or parts to the emergency relief scenario in the context sketched previously.

Leg I In the first part of the scenario, it is essential that for specific geographic areas, the UAV platforms should cooperatively scan large regions in an attempt to identify injured persons. The result of such a cooperative scan would be a saliency map pinpointing potential victims and their geographical coordinates and associating sensory output such as high resolution photos and thermal images with the potential victims. The saliency map could be then

used directly by emergency services or passed on to other UAVs as a basis for additional tasks.

Leg II In the second part of the scenario, the saliency map generated in Leg I would be used as a basis for generating a logistics plan for the UAVs with the appropriate capabilities to deliver boxes containing food, water, and medical supplies to the injured identified in Leg I. This would also be done in a cooperative manner among the platforms.

For the purpose of this thesis, we will focus on the second leg, which is an example of a logistics scenario. One approach to solving logistics problems is to use a task planner to generate a sequence of actions that will transport each box to its destination. Each action must then be executed by a UAV. By using a task planner instead of a special purpose solution, more flexibility is gained when planning to achieve several different goals and a more general solution is obtained.

This scenario provides several examples where knowledge processing middleware could be used to process data originally from sensors to lift it up to a level were deliberative and reactive functionalities could use it.

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

For example, to collect information about the current position of the boxes it might be necessary for the UAV to scan parts of the area for them. To detect and locate a box it might be necessary to take video streams from the onboard cameras and do image processing and anchoring, like in the traffic monitoring application. When the locations of the boxes have been established the information can be used to generate a domain model from which the task planner can generate a logistics plan.

Execution. Each plan operator in a plan generated by a task planner corresponds to one or more actions which a UAV has to execute. These actions can be considerably complex and require sophisticated feedback about the environment on different levels of abstraction. For example, for a UAV to follow a three dimensional path generated by a motion planner it is necessary to continually estimate the position of the UAV by fusing data from several sensors, such as GPS and IMU. Another example is when a UAV has lost its GPS signal due to malfunction or jamming and is forced to land using other techniques such as vision based landing. In this case, the UAV has to process the video streams from its cameras and for example look for a landing pattern which can be used to estimate the altitude and

position relative to the pattern. This information would then be used to safely land the UAV on the pattern.

Monitoring. Classical task planners are built on the fundamental assumption that the only agent causing changes in the environment is the planner itself, or rather, the system or systems that will eventually execute the plan that it generates. Furthermore, they assume that all information provided to the planner as part of the initial state and the operator specifications is accurate. This may in some cases be a reasonable approximation of reality, but it is not always the case. Other agents might manipulate the environment of a system in ways that may prevent the successful execution of a plan. Sometimes actions can fail to have the effects that were modeled in a planning domain specification, regardless of the effort spent modeling all possible contingencies. Consequently, robust performance in a noisy environment requires some form of supervision, where the execution of a plan is constantly monitored in order to detect any discrepancies and recover from potential or actual failures.

For example, a UAV might accidentally drop its cargo. Therefore it must monitor the condition that if a box is attached, it must remain attached until the UAV reaches its intended destination. This is an example of a *safety constraint*, a condition that must be maintained during the execution of an action or across the execution of multiple actions. A carrier can be too heavy, which means that it must be possible to detect take off failures where a UAV fails to gain sufficient altitude. This is called a *progress constraint*, where instead of maintaining a condition, a condition must be achieved within a certain period of time.

Describing and evaluating conditions like these based on the actions currently being executed is an important task for knowledge processing middleware. Chapter 7 describes how to use DyKnow to implement such an execution monitoring functionality, how to generate the necessary state sequences used as input, and how to integrate it with a task planner. By using execution monitoring it is possible to increase the robustness of the execution of plans generated by a classical task planner.

1.2 Knowledge Processing Middleware

Information and knowledge have traditionally been processed in tightly coupled architectures on single computers. The current trend towards more heterogeneous, loosely coupled, and distributed systems necessitates new methods for connecting sensors, databases, components responsible for fusing and refining information, components that reason about the system and the environment, and components that use the processed information. As argued in the introduction, there is a need for a principled and systematic framework for integrating these components and bridging the gap between sensing and reasoning in a physical agent. We therefore introduce the term knowledge processing middleware, defined as follows.

Definition 1.2.1 (Knowledge Processing Middleware) Knowledge processing middleware is a systematic and principled software framework for bridging the gap between the information about the world available through sensing and the knowledge needed when reasoning about the world. □

1.2.1 Design Requirements

Any proposed knowledge processing middleware must satisfy a number of requirements. The first requirement is that the framework 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 suitable forms to be used by reasoning functionalities*. In the traffic monitoring scenario, the primary inputs will consist of low level sensor data such as images, measurements from a barometric pressure sensor, coordinates from a GPS, laser range scans, and so on. However, there might also be high level information available such as geographical information and declarative specifications of traffic patterns and normative behaviors of vehicles. Knowledge processing middleware must be sufficiently flexible to allow the integration of all these different sources into a coherent processing system. Since the appropriate structure will vary between applications, a general framework should be agnostic as to the types of data and information being handled and should not be limited to specific connection topologies.

To continue with the traffic monitoring scenario, 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. Therefore a second requirement is the *support of quantitative and qualitative processing* as well as a mix of them.

A third requirement is that *both bottom-up data processing and top-down model-based processing should be supported*. Different abstraction levels are not independent. Each level is dependent on the levels below it to get input for bottom-up data processing. At the same time, the output from higher levels could be used to guide processing in a top-down fashion. For example, if a vehicle is detected on a particular road segment, then a vehicle model could be used to predict possible future locations, which could be used to direct or constrain the processing on lower levels. Thus, a knowledge processing framework should not impose a strict bottom-up data flow model nor a strict top-down model.

A fourth requirement is support for *management of uncertainty* on different levels of abstraction. There are many types of uncertainty, 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 it is not realistic to use a single approach to handling uncertainty throughout a middleware framework. Rather, it should allow many different approaches to be combined and integrated into a single processing system in a manner appropriate to the specific application at hand.

Physical agents acting in the world have limited resources, both in terms of processing power and in terms of sensors, and there may be times when these re-

sources are insufficient for satisfying the requests of all currently executing tasks. In these cases a trade-off is necessary. For example, reducing update frequencies would cause less information to be generated, while increasing the maximum permitted processing delay would provide more time to complete processing. Similarly, an agent might decide to focus its attention on the most important aspects of its current situation, ignoring events or objects in the periphery, or to focus on providing information for the highest priority tasks or goals. An alternative could be to replace a resource hungry calculation with a more efficient but less accurate one. Each trade-off will have effects on the quality of the information produced and the resources used. Another reason for changing the processing is that it is often context dependent and as the context changes the processing needs to change as well. For example, the processing required to monitor the behavior of vehicles following roads and vehicles which may drive off-road is very different. In the first case assumptions can be made as to how vehicles move which improves the predictive capability, while these would be invalid if a vehicle goes off-road. To handle both cases a system would have to be able to switch between the different processing configurations. A fifth requirement on knowledge processing middleware is therefore support for *flexible configuration and reconfiguration* of the processing that is being performed.

An agent should preferably not have to depend on outside help for reconfiguration. Instead, it should be able to reason about which trade-offs can be made at any point in time. This requires introspective capabilities. Specifically, the agent must be able to determine what information is currently being generated as well as the potential effects of changes it may make to the configuration of the processing. 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.

1.3 Thesis Outline

The thesis consists of four parts. The first part, Chapters 1–2, provides an introduction and a background to the thesis. The second part, Chapters 3–5, describes the details of our stream-based knowledge processing middleware framework DyKnow. The third part, Chapters 6–9, presents applications and extensions of DyKnow, which includes how the two example scenarios can be implemented and how to extend DyKnow to a multi-agent environment. The fourth part, Chapters 10–12, concludes the thesis with a discussion of DyKnow in a broader perspective as a fusion framework and in relation to other similar frameworks, a summary of the work presented, and a discussion about future work.

Chapter 2, Background, puts knowledge processing middleware into perspective by comparing it to existing general purpose middleware for distributed systems

and data stream management systems which extend traditional database technology with support for streams.

Chapter 3, Stream-Based Knowledge Processing Middleware, proposes a specific type of knowledge processing middleware based on the processing of asynchronous streams by active and sustained knowledge processes. Parts of this work have been presented in Heintz, Kvarnström, and Doherty (2008a,b).

Chapter 4, DyKnow, provides a formal description of DyKnow, a concrete stream-based knowledge processing middleware framework with a formal language for specifying knowledge processing applications. Earlier versions of this work have been presented in Heintz and Doherty (2004a, 2005a).

Chapter 5, A DyKnow CORBA Middleware Service, describes how DyKnow can be implemented as a CORBA middleware service.

Chapter 6, The UASTech UAV Platform, describes our UAV platform in enough detail to understand the rest of the chapters. This work has been presented in Doherty et al. (2004); Doherty, Kvarnström, and Heintz (2009).

Chapter 7, Integrating Planning and Execution Monitoring, describes how we have integrated planning and execution monitoring to implement parts of the emergency service scenario. The chapter shows how DyKnow can support the integration of sensing, acting, and reasoning by extracting information about the environment in order to facilitate monitoring the execution of plans. This work has been presented in Doherty, Kvarnström, and Heintz (2009); Kvarnström, Heintz, and Doherty (2008).

Chapter 8, Integrating Object and Chronicle Recognition, describes how we have integrated a chronicle recognition system and an object recognition functionality for anchoring object symbols to sensor data in order to implement a version of the traffic monitoring scenario using our UAV platform. This provides another concrete example of how DyKnow can be used to bridge the sense-reasoning gap. This work has been presented in Heintz and Doherty (2004b); Heintz, Rudol, and Doherty (2007a,b); Heintz (2001).

Chapter 9, DyKnow Federations, presents an initial approach to how DyKnow can be extended to facilitate multi-agent knowledge processing. The extension is illustrated by a multi-platform proximity monitoring scenario. This work has been presented in Heintz and Doherty (2008).

Chapter 10, Relations to the JDL Data Fusion Model, describes how DyKnow can support the functionalities on the different abstraction levels in the JDL Data Fusion Model, which is the de facto standard functional fusion model used today. This provides an argument that DyKnow is general enough to support a wide variety of applications which requires fusion and situational awareness. This work has been presented in Heintz and Doherty (2005b,c, 2006).

Chapter 11, Related Work, presents a selection of related agent architectures and robotics frameworks and discusses their support for knowledge processing in comparison to DyKnow.

Chapter 12, Conclusions, provides a concise summary of the work presented in the thesis and presents some interesting ideas as to how DyKnow could be further extended and applied in the future.

Chapter 2

Background

2.1 Introduction

In this chapter we put knowledge processing middleware into perspective by comparing it to existing general purpose middleware for distributed systems and to data stream management systems which extend traditional database technology with support for streams. Object-oriented, publish/subscribe, and event-based middleware approaches will be related to the requirements described in Section 1.2.1 and we argue that they fail to provide the necessary middleware support for knowledge processing applications. The same is argued for data stream management systems.

2.2 Middleware

As distributed applications become more common and more complex there is an increasing need to handle a diversity of components, underlying networks, and hardware. This requires sophisticated software support (Schantz and Schmidt, 2006). To counter this increasing complexity, a set of software frameworks called middleware has been developed. According to Emmerich (2000) "[m]iddleware resolves heterogeneity, and facilitates communication and coordination of distributed components." A classical example is CORBA (Object Management Group, 2008) which provides a framework for distributing objects between different platforms and languages, and a programming model where the physical location of an object is transparent to application programmers.

Middleware is expected to simplify the development of applications by hiding complexities and providing more appropriate interfaces on a higher level of abstraction. The following list is a quote describing desirable middleware features (Object Web, 2003):

- hiding distribution, i.e. the fact that an application is usually made up of many interconnected parts running in distributed locations;

- hiding the heterogeneity of the various hardware components, operating systems, and communication protocols;
- providing uniform, standard, high-level interfaces to the application developers and integrators, so that applications can be easily composed, reused, ported, and made to interoperate; and
- supplying a set of common services to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.

There are two aspects of middleware which are relevant for this thesis. The first is the distribution aspect where middleware hides the complexities of developing systems consisting of components running on different heterogeneous nodes in a network. The second aspect is that of providing higher level interfaces with appropriate support for the applications at hand. Since we are interested in supporting the development of knowledge processing applications it is important that the middleware provides the appropriate abstractions for working with those.

In general, middleware allows different components to interact by supporting some form of communication between the components. Middleware can be seen as the glue which holds a distributed application together.

2.2.1 Object-Oriented Middleware

A common and popular class of middleware frameworks developed for distributed systems is centered around the concept of an object. The idea is to provide a common object-oriented programming model disregarding the underlying network infrastructure, the physical location of objects, and the actual implementation language used. This class of middleware, which is the most mature since it has been around for more than 20 years, consists of for example CORBA (Object Management Group, 2008), Real-Time CORBA (Object Management Group, 2005), Ice (Henning, 2004), and Java RMI (Sun, 2000).

One particular category of object-oriented middleware, often called DRE middleware (Schmidt, 2002a,b), is specifically designed for distributed, real-time, and embedded environments. These middleware systems focus on issues such as increasing execution time predictability and reducing communication latency and memory footprint, making them particularly interesting for embedded knowledge processing applications on board a robotic system. However, pure DRE middleware frameworks do not by themselves satisfy the requirements for knowledge processing middleware.

Regarding the distribution aspect, the major weakness with object-oriented middleware is the use of an invocation-based client-server communication model. This means that each object reference needs to know which server hosts an object implementation, and each message must be sent directly to this server. Much of this is however hidden from application developers and taken care of by the object request broker. The main drawbacks of this design is that a server is a single point

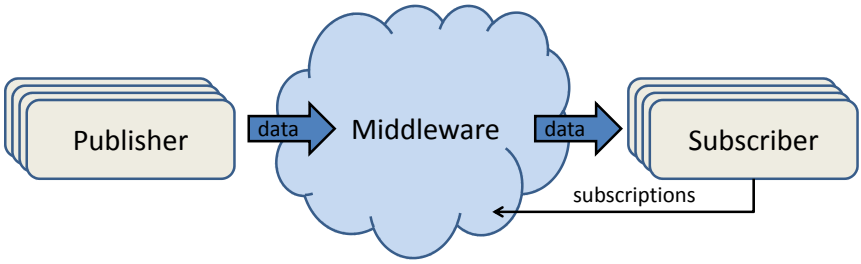


Figure 2.1: A conceptual overview of a publish/subscribe middleware.

of failure and that one-to-one communication does not scale very well for applications that need to disseminate the same information to many objects in a distributed system.

Object-oriented middleware such as CORBA usually provides a flexible way of creating new objects and sometimes even new interfaces at run-time. The downside is that all of these operations are procedural which makes it hard to reason about the current configuration. There is no support provided for declarative specifications of the required components. Therefore the requirement for flexible configuration and dynamic reconfiguration is not satisfied.

Finally, and perhaps most importantly, object-oriented middleware is very general and provides no specific support for creating representations on different levels of abstraction. However, it is of course possible to base knowledge processing middleware on object-oriented middleware. In fact, DyKnow is currently implemented as a service on top of CORBA.

2.2.2 Publish/Subscribe Middleware

Another important category of distribution middleware consists of those that use the *publish/subscribe pattern* of communication. A publish/subscribe system consists of a set of *publishers* publishing data and a set of *subscribers* subscribing to data (Figure 2.1). The publishers are also known as *producers* and the subscribers as *consumers*. Publish/subscribe middleware allows consumers to describe the data they are interested in through *subscriptions*. When a producer publishes new data the middleware delivers the data to each consumer with a matching subscription.

The purpose of this design is to provide a technology for transporting information between many producers and many consumers without them knowing about each other. Instead they share some common entity, like a communication channel or a topic of interest, which they interact with. The common entity will know about the producers and the consumers, but they will not know about each other.

A benefit of publish/subscribe systems is the support for many-to-many communication while decoupling consumers and producers. A consumer can transparently get information from many different producers over a single communication channel and a producer can reach many different consumers with a single oper-

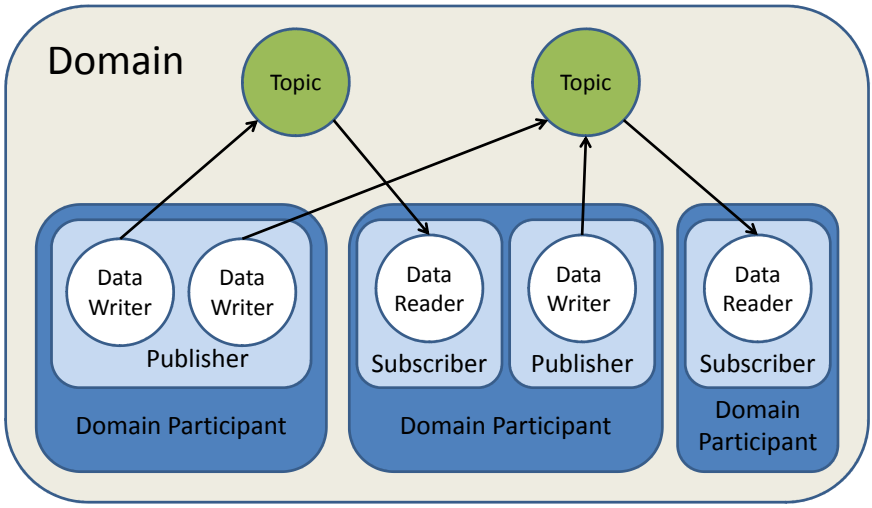


Figure 2.2: An overview of the components of the data distribution service.

ation. Another benefit is the possibility to add quality of service guarantees to different components in the publish/subscribe architecture.

Differences between implementations are mainly in the subscription specification languages they support and the underlying network communication used to do routing and dispatching. See Carzaniga, Rosenblum, and Wolf (1999) or Pietzuch (2004) for a survey.

The Data Distribution Service

An interesting version of the publish/subscribe concept is defined in the Object Management Group (OMG) standard for the data distribution service (DDS) (Object Management Group, 2007). It is a data-centric publish/subscribe distribution and communication infrastructure. Like all other publish/subscribe systems, DDS is designed to transparently distribute and share information between data producers and consumers.

Information produced and consumed is collected in *topics*, the main concept in DDS. A *domain* consists of a set of topics and a set of *domain participants* that read and write data to a topic (Figure 2.2). A domain participant can contain *publishers* and *subscribers*. A publisher writes data to a topic by using a *data writer* which acts as a proxy to a topic. A subscriber reads data from a topic by using a *data reader* which also acts as a proxy to a topic. If a domain participant wants to interact with more than one topic then a data reader or writer is required for each one.

What makes DDS special is its fine-grained quality of service policies. The following are some important examples:

- *Deadline* indicates the minimum rate at which a producer sends data or how

long a subscriber is willing to wait for new data.

- *Destination order* determines how a subscriber handles data that arrives in a different order than it is sent. It can either read the data in the order it is sent or in the order it is received.
- *Durability* specifies whether the data distribution service makes historic data available to subscribers that are added after the data has been sent.
- *Latency budget* is an optional guideline to be used by the system to implement optimizations in order to accommodate the subscribers' maximum acceptable latency.
- *Ownership* determines whether more than one publisher can publish data items to the same topic or not.
- *Ownership strength* determines which publisher is allowed to publish its data in the case of an exclusive ownership policy. This can be used to implement redundancy in a system.
- *Reliability* specifies whether or not a given subscriber will get the data reliably. If a data item is lost in a reliable channel the middleware will guarantee that it is resent.
- *Resource limits* specify how much local memory can be used by the middleware.
- *Time-based filter* provides a way to set a "minimum separation" period between data items, i.e. data items should not arrive more often than a certain period.
- *Transportation priority* is used to set the relative priority between different topics.

Since the concept of a many-to-many communication channel or stream is explicit in the design, the distribution service has a much better communication model for our purposes compared to object-oriented middleware. However it suffers from the same lack of suitable abstractions when it comes to knowledge processing. The available quality of service guarantees are very interesting and share several similarities with the stream policies defined in this thesis. Therefore the data distribution service is probably a good candidate to base knowledge processing middleware on. It would still be necessary to significantly raise the abstraction level to support knowledge processing on a suitable level of abstraction and provide a declarative specification to support flexible configuration and dynamic reconfiguration.

Event-Based Middleware

A special type of publish/subscribe middleware is event-based middleware where every piece of published data is seen as an *event* which subscribers can react to (Carzaniga, Rosenblum, and Wolf, 2001).

An important part of an event-based middleware is the capability to filter out events. There are two main approaches to filtering events, *topic-based* and *content-based*. In topic-based filtering a subscriber will subscribe to all events belonging to a particular topic. Content-based filtering, on the other hand, looks at the content of each event, which is often in the form of a set of attribute-value pairs. All events whose attribute values satisfy the filter are matched by the subscription. The subscriptions in the data distribution service is a good example of topic-based filtering. The data distribution service also supports content-based filters, but only applies these to data within a particular topic.

Two interesting event-based publish/subscribe services are the CORBA real-time event service (Harrison, Levine, and Schmidt, 1997) and the CORBA real-time notification service (Gore et al., 2004; Gruber, Krishnamurthy, and Panagos, 2001). These services provide a basic publish/subscribe architecture where a channel is implemented as a CORBA object which publishers and subscribers can connect to in order to send and receive events. The real-time event service subscriptions are topic-based while the notification service also supports content-based subscriptions. The real-time event service has a very limited vocabulary with respect to filtering of events, where you can only filter based on the type and the source of an event as represented by two integers. The notification service on the other hand has a more complex language for expressing filter constraints called the extended trader constraint language.

When information is seen as events, it is natural to think about how to specify and detect complex composite events in a stream of events. This line of research has resulted in several languages for expressing composite events, including Siena (Carzaniga, Rosenblum, and Wolf, 2001), Jedi (Cugola, Nitto, and Fuggetta, 2001), Gryphon (Banavar et al., 1999), and Elvin (Segall and Arnold, 1997).

Recently the scope of this research has been extended to also include the creation of new events based on events detected so far. This is often called complex event processing or event stream processing (Luckham, 2002).

Event-based middleware provides another step in the direction of knowledge processing middleware since it provides concepts and tools to define and detect complex events. Event specifications are often made in a declarative language making it possible to reason about them. However, the abstraction is still limited to expressing information about events. There are for example no abstractions for talking about continuous variables or objects. Therefore event-based middleware is limited to applications which can be expressed in terms of event processing. Even though this is a large and general class of applications we believe that there are other abstractions which are essential for knowledge processing applications.

2.3 Data Stream Management Systems

In the nineties, the database community realized the need for managing streaming data, as opposed to data stored in tables, in order to handle massive amounts of real-time data. This data can be generated from different types of logs, including web servers and network surveillance systems, or from sensor networks producing

data at a high rate. The key observation is that in order to be able to keep up with the pace, a data management system can not afford to store data in tables but has to process it on-the-fly as each tuple becomes available (Abadi et al., 2003; Babcock et al., 2002; Motwani et al., 2003; The STREAM Group, 2003).

From a database perspective, a data stream management system stores and queries streams of data instead of tables of data. The supported query language is often based on SQL. The stream-based queries are sometimes called continuous queries since they have to be continuously evaluated as new tuples become available. The main research issues are continuous query languages and the optimization of continuous queries. A major difference compared to the stream-like functionality provided by publish/subscribe systems is that a continuous query in a data stream management systems actually transforms streams as opposed to only describing what elements are requested.

These data stream management systems are not really middleware since they are usually not distributed. Some form of middleware is therefore required to transport the streams to and from a data stream management system. The functionality they provide, however, bears resemblance to the middleware functionality that we are striving after.

Most data stream management systems at least partially fulfills many of the requirements for knowledge processing middleware. By providing a declarative language where the processing of streams are described the sixth requirement is met. Since all data stream management systems allow queries to be added and removed at run-time they also support the requirement for flexible configuration and reconfiguration. However, like the middleware approaches described earlier no explicit support is provided for lifting the abstraction level. Since most data stream management systems are monolithic systems hosted on a single computer they usually do not provide much support for the integration of information from distributed sources.

2.4 Summary

This chapter has presented some existing middleware approaches and data stream management systems which could be used to support the implementation of a solution to bridging the sense-reasoning gap. One common feature is that they all are very general and only consider data, not higher level abstractions which are necessary for knowledge processing applications. The middleware approaches all provide adequate support for the integration of information from distributed sources requirement, especially those which are based on the publish/subscribe communication pattern.

At the same time data stream management systems provide an appropriate data model where incremental streams of data are continually processed by continuous queries. As will be shown in the next chapter, we propose to base knowledge processing middleware on a similar concept of streams.

The conclusion is that even though none of the described middleware approaches themselves provide the necessary support for knowledge processing middleware

they could all be used as a foundation providing low-level support for communication and distribution of processing. As will be seen in Chapter 5, DyKnow is currently implemented as a CORBA service which uses the notification service to provide a publish/subscribe communication model.

Part II

Knowledge Processing Middleware

Chapter 3

Stream-Based Knowledge Processing Middleware

3.1 Introduction

In the first chapter we presented a set of requirements that are necessary or desirable for any form of knowledge processing middleware. How these requirements should be satisfied was intentionally left open. In this chapter we propose the use of *stream-based* knowledge processing middleware as one appropriate basis for satisfying the requirements.

As exemplified by the traffic monitoring scenario (Section 1.1.1 on page 5), physical agents tend to require a great deal of processing of information and knowledge at different levels of abstraction. This processing is generally separated into a number of distinct and well-defined functionalities, which we model as active and sustained *knowledge processes*. For example, image processing and helicopter state estimation may be modeled as two distinct knowledge processes with clearly defined responsibilities as can every other node in Figure 1.2 on page 6.

Most knowledge processes need information generated by other processes, and in turn generate refined information that may be required by others. In order to properly model the flow of information, it is essential for a framework to take into account the dynamic nature of knowledge processing, where information only becomes available incrementally. For this reason, we model the information flow in terms of *streams* of information flowing between knowledge processes. For example, every arrow in Figure 1.2 could be realized as a stream.

A *stream-based knowledge processing application* is a network of knowledge processes connected by streams. Both the input and the output of a knowledge process is in the form of streams. An object recognition process could for example take a stream of images as input and provide streams of recognized objects as output. Since many processes might be interested in the information produced by a knowledge process, output streams are made available through *stream generators*. A process that is interested in the output of another process can *subscribe* to the

associated stream generator, which creates a new stream connecting the two processes. Each subscription is associated with a *policy* which specifies the desired properties of the created stream. To satisfy the policy it might be necessary for the stream generator to filter out elements or even to add new approximated elements to the stream.

In the remainder of this chapter, we define the concept of stream-based knowledge processing middleware in more detail. The definitions in this chapter are intentionally general to capture the nature of any stream-based knowledge processing middleware. The exact formal definitions may vary between instantiations. In the next chapter a specific instantiation called DyKnow is described where the definitions are made formal.

3.2 Stream

As noted above, knowledge processing for a physical agent is fundamentally incremental in nature. A knowledge process that produces information for a given stream does so one piece at a time, in real time. The receiving process can choose when to process this piece of information, but this can only be done after the information has become available. For example, a knowledge process computing position estimations from GPS readings can not compute a new estimate until it has received the latest GPS coordinate.

From an implementation point of view, this property will automatically be satisfied. Retrieving information before it is produced is logically impossible. However, we also require a formal model of streams and knowledge processes. Such a model should preferably provide a way to completely describe not only a snapshot of a system but its history over time. This is, for example, essential for the ability to validate an execution trace relative to a formal system description.

Given this requirement, we choose to view a stream as a structure containing its own history over time. In other words, for any time-point, it is possible to determine which elements are available in a stream at that time-point.

Definition 3.2.1 (Stream) A *stream* is a set of *stream elements*, where each stream element is a tuple $\langle t_a, \ldots \rangle$ whose first element, 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 for the stream. □

Since the available time is unique the tuples in a stream can be totally ordered by their available times and the stream can be seen as a sequence of stream elements. To guarantee that there is a total order, a $<$ operator must be defined on the time-point domain. We will see a stream either as a set or as a sequence, which ever is most appropriate.

Streams provide a useful and natural model of the incremental nature of knowledge processing and of the availability of information. That the content of a stream becomes incrementally available means that at any time only a prefix of the stream is available. At time-point t only those elements with an available time less than or

equal to t are available. This means that at time-point t a stream can be seen as the sequence of those elements with an available time not greater than t.

In order for a process to access the output of another process it must be able to refer to a stream generator. Each stream generator is therefore associated with a *label*. Though a label could be opaque, it often makes sense to use structured labels. For example, given that there is a separate position estimator for each car, it makes sense to provide an identifier i for each car and to denote the (single) stream generator of each position estimator by $position[i]$. Then, it is sufficient to know the car identifier to generate the correct label.

Asynchronicity. In general, streams are asynchronous which means that it is not possible to predict when the next element will be available in a stream given the currently available prefix. For example, the information can be inherently asynchronous, such as a stream representing the pressing of a button or the detection of reckless overtakes. It could also be that even if the information is synchronously sampled with a certain period, unpredictable delays in a system can cause a stream to be asynchronous.

Due to the asynchronous nature of streams it is essential that stream-based knowledge processing middleware implementations have support for notifying the receiving processes when new elements become available. This means that when a new element is made available in a stream it should be pushed to the connected processes as soon as possible. The alternative would be for a process to use polling to get the elements of a stream. However, polling is inefficient and it risks losing information if a stream does not buffer enough elements. The more frequent the polling is, the more inefficient it is. A push-based stream system also lends itself easily to "on-availability" processing, i.e. processing data as soon as it becomes available, and to the minimization of processing delays, compared to a query-based system where polling introduces unnecessary delays in processing.

Decoupling. A benefit of using streams is that they allow knowledge processes to be decoupled from each other since they easily support a *publish/subscribe* communication mechanism. Any information generated by a knowledge process is published using a stream generator. A knowledge process interested in this information can subscribe to it, which creates a new stream connecting the two processes. Note that it would also be possible to allow many stream generators to contribute content to a single stream. Streams and stream generators thereby support many-to-many communication between consumers and producers who do not need to know about each other. This also means that a knowledge process can publish new elements when they are computed and then continue its processing without having to wait for all the subscribers to receive and process the element. In fact, the delivery of the elements in a stream is completely independent of the generation of the elements.

Decoupling knowledge processes generating information from consumers of the information, such as other knowledge processes, has many benefits. Using a publish/subscribe pattern of communication decouples knowledge processes in

time, space, and synchronization (Eugster et al., 2003). This separation provides a very good foundation for supporting distributed knowledge processing applications.

First, consumers and producers only have to know about the stream generators they use, they do not have to know about each other. Since each stream generator is referred to by a label it is enough for a consumer to know the label of the desired stream generator to access it. Who produces an element for a stream generator is not important to a consumer and who receives the element is not important to a producer. This can, for example, be used to add redundancy to a system by having several producers provide the same information. It also supports the distribution of processes on different nodes in a distributed system since a consumer does not need to know where a producer is hosted and vice versa.

Second, the generation of information is separated from its delivery. A knowledge process is therefore not blocked even if one of the subscribers is either temporarily unavailable or takes a long time to process the information. This also allows the input and output streams of a knowledge process to have different update rates. For example, a knowledge process estimating the position of an agent can use an input stream with a higher update frequency than its output stream in order to reduce the uncertainty in the estimated positions.

3.2.1 Policy

A stream can have many different properties including different types of quality of service guarantees. Different consumers will have different requirements on the properties of a stream for its content to be useful to them. For example, the chronicle recognition engine described in Chapter 8 requires that elements arrive in the order in which they are produced. Another requirement could be that a stream should only contain elements which constitute a significant change compared to the previous element in the stream. Other consumers might require updates with a certain frequency or elements which are not delayed by more than a constant amount of time.

To specify the desired properties of a stream a *policy* is used. Examples of properties are the duration of a stream, the maximum allowed delay, and the sample period of a stream. A policy can also specify advice for how to generate a stream that satisfies the properties. An example of advice is how to handle the case when there is a missing element or when an element whose delay is too high is received. When a process subscribes to a stream generator the subscription includes a policy. Given this policy it is up to the stream generator to generate a stream which satisfies it. The policy allows the process to specify the desired properties and also to guide the generation of the stream. For introspection purposes, policies should be declaratively specified.

Definition 3.2.2 (Policy) A *policy* is a declarative specification of the desired properties of a stream which may include advice on how to generate the stream. □

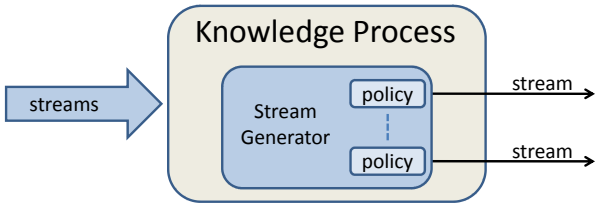


Figure 3.1: A knowledge process operates on streams to generate streams according to policies.

3.3 Knowledge Process

We model computations and processes within the stream-based knowledge processing framework as active and sustained *knowledge processes*. The complexity of such processes may vary greatly, ranging from simple adaptation of raw sensor data to image processing algorithms to potentially reactive and deliberative processes.

Definition 3.3.1 (Knowledge process) A *knowledge process* is an active and sustained process whose inputs and outputs are in the form of streams. □

A knowledge process operates on streams. It can either take streams as input, produce streams as output, or both. Each input stream to a knowledge process is specified by a policy. If a knowledge process generates output, then it is made available through a stream generator since there might be many processes interested in the output but with different requirements on it. Another process can then subscribe to the stream generator, which will generate a stream by adapting the output of the knowledge process according to a policy. For example, given a process estimating the position of an agent every 100 ms, a stream with any sample period which is a multiple of 100 ms could be provided. The stream generator provides the functionality to make the necessary adaptation. This separates the generation from the adaptation of stream content. An abstract view of a knowledge process is shown in Figure 3.1.

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

3.3.1 Primitive Process

Primitive processes serve as interfaces to the outside world, connecting to sensors, databases, or other information sources and provide their output in the form of streams. Such processes have no stream inputs but provide a non-empty set of stream generators. A conceptual primitive process is shown in Figure 3.2. Often they tend to be quite simple, mainly adapting data in a multitude of external representations to the stream-based framework. For example, one process may use

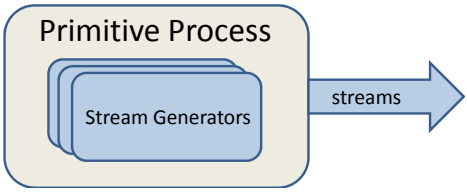


Figure 3.2: A conceptual primitive process.

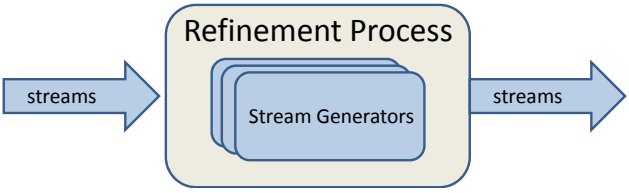


Figure 3.3: A conceptual refinement process.

a hardware interface to read a barometric pressure sensor and provide a stream generator for this information. However, greater complexity is also possible, with primitive processes being responsible for tasks such as image processing.

Definition 3.3.2 (Primitive process) A *primitive process* is a knowledge process which does not take any streams as input but provides output through one or more stream generators. □

3.3.2 Refinement Process

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 the output as streams. A conceptual refinement process is shown in Figure 3.3. A refinement process can basically do any processing imaginable on its input streams. It could for example implement different kinds of sensor processing such as image processing extracting objects from a video stream, fusion of sensor data such as Kalman filters estimating the position of a robot from GPS and IMU data, or reasoning about the qualitative spatial relations between objects.

Definition 3.3.3 (Refinement process) A *refinement process* is a 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. If a concrete realization of stream-based knowledge processing middleware allows a pro-

cess to change the policies of its inputs during run-time, then a refinement process can dynamically tailor its subscriptions depending on the streams it is supposed to create. For example, if a position estimation process is supposed to compute the position of a robot with 10 Hz then it could either subscribe to its inputs with the same frequency or choose to subscribe with a higher frequency in order to filter out more noise. If the subscriptions to the stream generators of a process change, the process might have to change its subscriptions as well.

In certain cases, a process is first required to 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 has consequences for the policies that a stream generator for the output can support. If it takes 30 seconds to collect data and to initialize the internal state of the knowledge process then the stream generator could not accept subscriptions requiring data earlier than 30 seconds after the creation of the knowledge process. This can be remedied if the process is able to request 30 seconds of historic data when it is created.

3.3.3 Configuration Process

In many cases, it is beneficial to use separate knowledge processes to handle information related to specific objects. Since we are interested in describing systems that dynamically create and destroy objects, there is also a need to dynamically add and remove knowledge processes and streams related to those objects. For example, in the traffic monitoring scenario there is a need to process streams containing information about a varying number of cars. New cars may be detected and old cars forgotten. Using only refinement processes, a solution is to have a stream containing the information about all cars. Even if this is possible, and sometimes even desirable, it has several downsides.

For example, instead of containing car states the stream would contain sets of car states. Now assume that we have observed 100 cars and we are currently tracking one of them. To track this car we need an updated position estimation every 100 ms, but for the rest of the cars it is enough with an update every second. If we have a single stream containing the position of all cars then we either have to update all cars every 100 ms or introduce a more expressive policy which allows different cars within a single stream to have different sample rates. This is an unnecessary complexity which can be avoided by allowing a flexible number of processes and streams.

To handle these issues and to make the control of the system more fine-grained a *configuration process* is introduced. It takes a set of streams as input but does not have any output streams. Instead, it enables dynamic reconfiguration by adding or removing streams and processes.

Definition 3.3.4 (Configuration process) A *configuration process* is a knowledge process which takes streams as inputs, has no stream generators, and creates and removes knowledge processes and streams. □

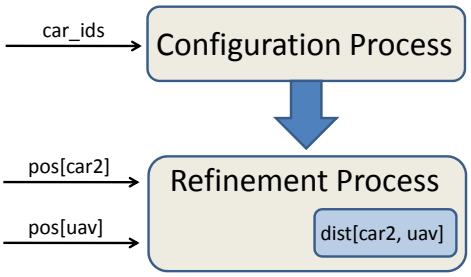


Figure 3.4: A configuration process where the current element of car_ids is {car2}.

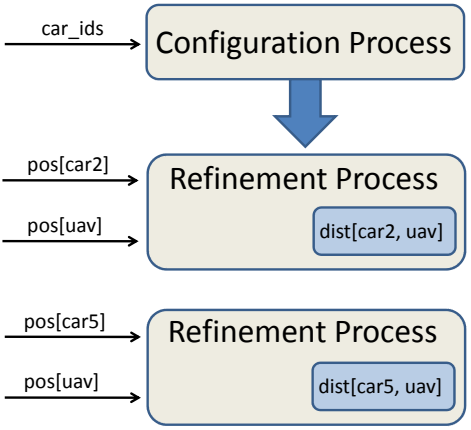


Figure 3.5: A configuration process where the current element of car_ids is {car2, car5}.

For example, assume there is a knowledge process which produces a stream containing the set of identifiers of all cars currently being tracked. Each time the system starts tracking a new car its identifier is added to the set and each time the system loses track of a car its identifier is removed from the set. A configuration process could take this stream of sets of identifiers as input and ensure that for each identifier in the set there is a knowledge process estimating the distance from the car to the UAV. When a new identifier is added to the set, a new knowledge process is created, and when an identifier is removed from the set, the corresponding knowledge process is removed. The configuration process would capture a correspondence between the content of its input streams and the knowledge processes in the system. An instantiation of the example is shown in Figures 3.4 and 3.5.

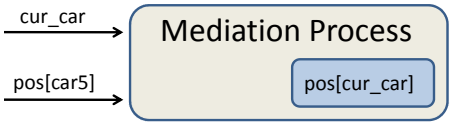


Figure 3.6: A mediation process where the current element of cur_car is car5.

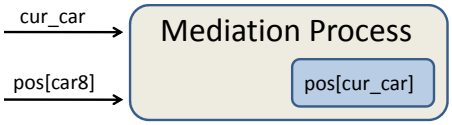


Figure 3.7: A mediation process where the current element of cur_car is car8.

3.3.4 Mediation Process

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

Definition 3.3.5 (Mediation process) A *mediation process* is a knowledge process that changes its input streams and mediates the content on the varying input streams to a fixed number of stream generators. □

There are at least three cases when a mediation process is needed.

First, when the set of inputs to a knowledge process changes over time. For example, if the position of each observed car is provided in a separate stream then the set of such streams changes over time. To create a stream containing the positions of all cars we need to merge the content of each of these streams. This would easily be done with a refinement process if the exact number and identifiers of the input streams were static and known from the start, but this is not always the case. Instead we would like to specify a process which takes all streams containing car positions as input, where the set of cars is given by another stream. Taking a stream of sets of car identifiers and mediating the content from the position stream for each of these car identifiers would be an example of a mediation process.

Second, when the inputs are not known at specification time. For example, if we would like to create a knowledge process which can take inputs from different sources such as a sensor on a robot, a simulator, or a log file then it might not be known in advance which source will be used. Instead of making a guess we can add a mediation process which will switch between the different sources and provide a single static stream generator independently of the actual source. It would also be possible to change the source at run-time.

Third, when the content of all input streams is not needed all the time. For example to create a stream of the position of the currently tracked car. There might

be many cars in the system, but only one is currently tracked. Instead of subscribing to all car position streams, it is enough to subscribe to the one stream which is needed. This is mainly an optimization, but in systems with a large number of streams this might make a major difference.

It is often useful to describe the inputs to a mediation process using a structured label where one or more of its arguments are not instantiated but rather determined by the input on one or more of the input streams to the mediation process. For example, if there is a stream cur_car which contains the identifier of the car which is currently being tracked, then a mediation process could take this stream as input and instantiate the structured label position[*car*], where *car* is replaced with the currently tracked car identifier. When the instantiation of this structured label changes, the inputs to the mediation process are updated accordingly. An example is given in Figures 3.6 and 3.7.

3.3.5 Stream Generator

A *stream generator* is used to adapt the output of a knowledge process to create many different streams. This allows the content generated by, for example, a sensor or a process to be used to create many streams each with different properties. Assume there is a process estimating the position of an agent every 100 ms. A stream generator for this process could provide streams with any update period which is a multiple of 100 ms. It might also be able to generate a stream containing an element each time the position has changed significantly according to some measure.

Definition 3.3.6 (Stream generator) A *stream generator* is a part of a knowledge process which generates streams according to policies from output generated by the knowledge process. □

One reason to introduce a stream generator, which also processes streams, is to separate the creation of content of a stream and the creation of different streams by adapting this content. For example, in the case where a stream sampled with 1 Hz should be created from one sampled with 10 Hz, we could have had a knowledge process taking the stream containing 10 samples a second and producing a stream of 1 sample a second. However, if a second stream with a sample period of 5 samples a second were requested then another knowledge process would be required. Instead of duplicating the knowledge process a special stream adaptation process is used, the stream generator. By having a single knowledge process with a stream generator capable of creating both streams the application is simplified.

Another reason to introduce a stream generator is to separate the processing of streams from the creation of streams. Since a stream generator is only a promise to create streams, if requested by a subscription, no content has to be produced unless there is at least one subscription. The stream generator also acts as an interface to the knowledge process which determines the properties of the processing. This allows fine-grained control by restricting the processing to what is necessary to satisfy the current subscriptions. For example, if the only subscription to the stream

generator is for a stream sampled every second, then the process only has to do whatever is necessary to provide an output every second. In some cases it would be enough to actually sample the inputs to the process every second. In other applications a higher sample frequency might be needed. In any case, the process can optimize its processing and its inputs based on the current subscriptions.

While it should be noted that not all processing is based on continuous updates, neither is a stream-based framework limited to being used in this manner. For example, a path planner or task planner may require an initial state from which planning should begin and usually cannot take updates into account. Even in this situation, decoupling and asynchronicity are important, as is the ability for lower level processing to build on a continuous stream of input before it can generate the desired snapshot. The stream generator interface to the stream generated by a knowledge process ought therefore to also support snapshot queries. These queries would be answered relative to the stream produced by the knowledge process.

A policy should act as a specification for how a stream generator should generate a stream in order for the content to satisfy the policy. For example, even if a stream's elements should be ordered according to some criteria they could be made available to the stream generator in a different order. If the maximum allowed delay for the stream allows it, then the stream generator could wait for the missing elements to be available. If that is not an option then the stream generator either has to find another way to satisfy the policy or the policy will be violated. How to handle these cases is implementation dependent and allows for a large variation of solutions.

3.4 Summary

A stream-based knowledge processing application consists of a number of knowledge processes connected by streams. A knowledge process has stream generators which make the produced output available in the form of streams. A stream generator can be subscribed to by an arbitrary number of processes. A subscription can be viewed as a continuous query, which creates a distinct asynchronous stream connecting the two processes onto which new data is pushed as it is generated. A subscription includes a policy which specifies the properties of the stream, which should guide the creation of it. A policy could for example limit the maximum allowed delay, the time between two elements in the sequence, and the order of the elements in the stream.

Four different types of knowledge processes are identified. A primitive process allows a process that provides information to be integrated with a knowledge processing application by making the information available in the form of streams. A refinement process takes streams and computes more refined streams by for example merging, filtering, abstracting, and correlating its inputs. A mediation process mediates content from many streams into a single stream by dynamically changing its inputs depending on the content of one or more of its input streams. Finally, a configuration process controls the configuration of the knowledge processing application by continually adding and removing processes and streams. Together

they cover a very wide variety of knowledge processing applications.

Since knowledge processes are active and they are decoupled from each other through the use of streams, stream-based knowledge processing middleware is very well suited for distributed applications.

Chapter 4

DyKnow

4.1 Introduction

So far, we have talked about stream-based knowledge processing middleware in general. For example, in the previous chapter we did not exactly define what information a stream may contain or the properties of a stream that a policy can specify. In this chapter we will present one concrete stream-based knowledge processing middleware framework called DyKnow[1].

DyKnow provides both a conceptual framework for modeling knowledge processing and an implementation infrastructure for knowledge processing applications. This chapter focuses on the formal framework while the next chapter describes the implementation infrastructure. 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.

We start by presenting the ontology of the formal framework. It defines *objects* and *features* which for example may represent attributes of these objects. These are the two types of entities that DyKnow can explicitly model knowledge about. Since we are modeling a dynamic world a feature may change values over time. Then, the DyKnow knowledge processing domain is presented, which formally defines streams and knowledge processes. A stream that represents an approximation of the value of a feature over time is called a *fluent stream*. A fluent stream is a specialization of the general stream concept introduced in the previous chapter. 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 or computational unit provides a *fluent stream generator* which creates fluent streams from the output of the corresponding knowledge process according

[1] DyKnow stands for dynamic knowledge processing. Some of our papers state that DyKnow stands for "dynamic knowledge and object processing", but it is more appropriate to see objects as one type of entity to process knowledge about rather than to see 'knowledge' and 'object' as two distinct entities on the same level.

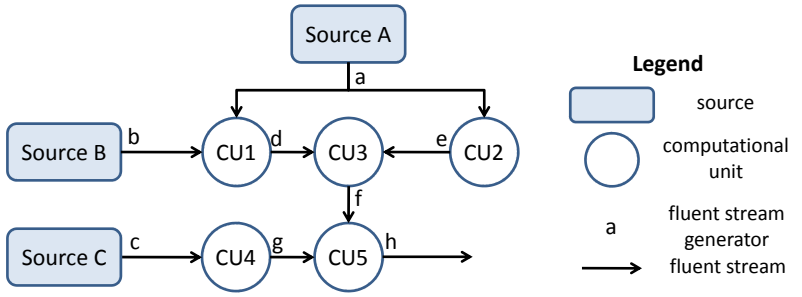


Figure 4.1: A conceptual DyKnow knowledge processing application consisting of sources and computational units producing fluent streams.

to *policies*. A conceptual DyKnow application is shown in Figure 4.1. Finally, we define the DyKnow knowledge processing language KPL which is the declarative language used by DyKnow for specifying knowledge processing applications.

4.2 Ontology

An ontology defines the types of entities that make up the world. For modeling purposes, DyKnow views the world as consisting of two types of entities: *Objects* and *features*. They are used in DyKnow to model the environment of an agent. We do not claim that these entities actually exist in the world, only that they are appropriate when describing it.

4.2.1 Object

The world is viewed as consisting of a set of distinct *objects*. No difference is made between concrete and abstract objects. In the UAV domain, each UAV may be viewed as an object as well as each blob found by an image processing system, and each car hypothesized as existing.

4.2.2 Feature

Properties of the world are called *features*. A feature could for example be an attribute of an object, such as the position of a car, or a relation between objects, such as whether two cars are beside each other or not. A feature has a well-defined value at every time-point. Since the world is dynamic, this value may change over time. For example, a car has a well-defined position at each time-point but the position may change if the car is moving.

4.3 Knowledge Processing Domain

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. The syntax of the knowledge processing specification language is described in Section 4.4 and the semantics in Section 4.5.

Definition 4.3.1 (Knowledge processing domain) A *knowledge processing domain* is defined by 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. The temporal domain T must be associated with a total order ($<$) and functions for adding ($+$) and subtracting ($-$) time-points. □

From a knowledge processing domain $D = \langle O, T, P \rangle$ the set of all possible values V_D, the set of all possible samples S_D, the set of all possible fluent streams F_D, the set of all possible sources R_D, and the set of all possible computational units C_D will be defined. When the domain D is understood from the context, the subscript D will be left out when referring to these sets.

Example 4.3.1 (Domain) The example knowledge processing domain used in the rest of this chapter is defined by the tuple $\langle \{o_1, \ldots, o_{10}\}, \mathbb{Z}^+, P \rangle$, where $P = \{p_1, \ldots, p_{10}\} \cup \{s_1, \ldots, s_{10}\}$ is the set of primitive values (positions and speeds). The set T of time-points is the set $\mathbb{Z}^+$ of non-negative integers including zero. □

4.3.1 Value

Values are used to construct all the other derived elements of a domain such as fluent streams and computational units. A value is either a simple value, the constant `no_value`, or a tuple of values. This recursive structure is necessary to be able to represent highly structured values such as states representing the values of a collection of features. No domain $D = \langle O, T, P \rangle$ may contain `no_value` as a member in the sets O, T, or P. The intended use of `no_value` is to represent that a fluent stream has no value at a particular time-point.

Definition 4.3.2 (Simple value) Let $D = \langle O, T, P \rangle$ be a knowledge processing domain. A *simple value* in D is either an object constant from O, a time-point from T, or a primitive value from P. The set of all possible simple values in a domain D is denoted by W_D. □

Definition 4.3.3 (Value) Let D be a knowledge processing domain. A *value* in D is recursively defined as follows: A simple value in W_D is a value, the constant `no_value` is a value, and if $v_1, \ldots, v_n$, $n \geq 0$, are values then $\langle v_1, \ldots, v_n \rangle$ is a value. The set of all possible values in a domain D is denoted by V_D. □

Example 4.3.2 (Value) Examples of simple values from the domain from Example 4.3.1 are p_4, o_3, and s_9. An example of a non-simple value is the tuple $\langle \texttt{no\_value}, \langle o_{10} \rangle \rangle$. □

4.3.2 Fluent Stream

Though we would like to have access to the actual value of a feature over time, these values will usually not be completely known by an agent. There are for example inherent limitations in the sensing and in the processing which affects the accuracy of the available information. Instead we have to create approximations.

In DyKnow, an approximation of the value of a feature over time is represented by a *fluent stream*. A fluent stream is a stream of *samples*, where each sample is a stream element which represents an observation or an estimation of the value of the feature at a particular time-point called the *valid time*. The concept of valid time is similar to the one used in temporal databases, where "[t]he valid time of a fact is the time when the fact is true in the modeled reality" (Jensen et al., 1998).

Any realistic knowledge processing application must take into account the fact that both processing and communication takes time, and that delays may vary, especially in a distributed setting. This means that a sample with a valid time t will not necessarily be available at time-point t. Each sample is therefore tagged with its *available time*, which corresponds to the time when the sample is available in the fluent stream. A sample is available when it has passed through all communication channels and is ready to be processed by the receiving process.

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 has not yet arrived. The valid time and the available time are independent of each other.

There are at least two reasons for separating valid time from available time.

First, several estimations of the value at a specific time-point can be made for the same feature because we have a non-monotonic system where the best available information about a time-point may vary. For example, a knowledge process could very quickly provide a first rough estimate of some feature, after which it would run a more accurate and time consuming algorithm, and eventually provide a better estimate. The different estimations would have the same valid time but different available times.

Second, it allows us to model delays in the availability of the value. The delay is the difference between the available time and the valid time, the greater the difference the greater the delay. The delay could for example be caused by processing of the value. If the value is not delayed then the available and valid times are the same.

Example 4.3.3 (Valid and available time) If a picture is taken at time-point t and arrives at a knowledge process at time-point t' then the valid time of the picture is t and the available time is t'. If a blob is then extracted from the picture by the knowledge process, then the valid time of the blob is still t but the available time is the time when the extracted blob has been received by a subscribed process. □

Definition 4.3.4 (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 the *available time* of the sample, $t_v \in T$ is the *valid time* of the sample, and $v \in V_D$ is the *value* of the sample. The set of all possible samples in a domain D is denoted by S_D. □

A sample $\langle t_a, t_v, v \rangle$ can be used to represent the fact that the estimated value of a feature at a particular time-point t_v is v and that this estimation is available to a process at t_a.

Example 4.3.4 (Sample) Assume a knowledge process takes a fluent stream g as input and generates a new fluent stream h, where g and h represent two different approximations of a feature f. The knowledge process could for example be a filter which takes a stream of measurements of f and computes the weighted average of several samples in order to filter out noise. If g contains a sample s_1 with valid time 10 and available time 11 then this would represent that g contains an approximation of f at time-point 10 which is available to the process at time-point 11. Assume the processing of the sample by the knowledge process takes three time units and that communicating this sample to the receiving process takes two time units. The resulting sample s_2 would then have the valid time 10, since it is still an approximation of f at time-point 10, while the available time would be 16. The first sample would be represented by the tuple $\langle 11, 10, v \rangle$ and the second sample by the tuple $\langle 16, 10, v' \rangle$. □

After having introduced samples we can now define a fluent stream as a stream containing samples.

Definition 4.3.5 (Fluent stream) A *fluent stream* in a domain $D = \langle O, T, P \rangle$ 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. □

The constant `no_sample` can never be part of a fluent stream. Instead it is used to represent that a query to a fluent stream does not have an answer. For example, suppose a fluent stream g represents an approximation of the value of the feature f over time. If g is queried about the value of f at time-point t, and it contains no sample with valid time t, then the query can return `no_sample` to indicate this lack of information. For two examples, see Definitions 4.3.8 and 4.3.12 below.

Note that any set of samples having unique available times corresponds to exactly one sequence of samples totally ordered by their available times, and vice versa. Therefore, any fluent stream $\{\langle t_{a_1}, t_{v_1}, v_1 \rangle, \ldots, \langle t_{a_n}, t_{v_n}, v_n \rangle\}$ can be written as a sequence $[\langle t_{a_1}, t_{v_1}, v_1 \rangle, \ldots, \langle t_{a_n}, t_{v_n}, v_n \rangle]$ where $t_{a_i} < t_{a_{i+1}}$ for each $i < n$. We will use both the set and the sequence notation.

Example 4.3.5 (Fluent stream) The sets $f_1 = \{\langle 1, 1, v_1 \rangle, \langle 3, 2, v_2 \rangle, \langle 4, 5, v_3 \rangle\}$ and $f_2 = \{\langle 2, 1, v_4 \rangle, \langle 4, 1, v_5 \rangle, \langle 5, 1, v_6 \rangle\}$ are fluent streams. A visualization of the two fluent streams is shown in Figure 4.2. □

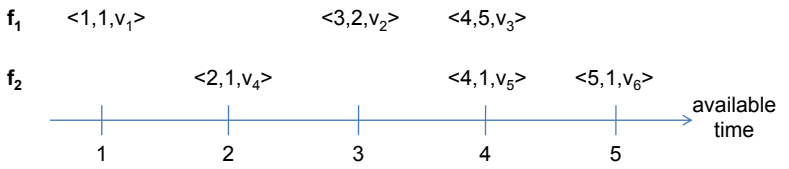


Figure 4.2: Two example fluent streams.

To simplify later definitions a number of utility functions will now be defined.

To access the parts of a sample, three functions *atime*, *vtime*, and *val* are introduced.

Definition 4.3.6 Let D be a domain and $s = \langle t_a, t_v, v \rangle$ be a sample in S_D. Then, $atime(s) \stackrel{\text{def}}{=} t_a$, $vtime(s) \stackrel{\text{def}}{=} t_v$, and $val(s) \stackrel{\text{def}}{=} v$. □

Due to the incremental nature of a fluent stream, which is modeled by the available time, not all samples in a fluent stream are considered available at all times. That a sample is not available could represent that it has not yet been generated by the system. For example, in an implemented system a measurement made by a sensor will not be available to any process until it is actually measured. To refer to the available samples at a particular time we introduce the function $available(f, t)$.

Definition 4.3.7 (Available samples) The function $available(f, t)$: $F \times T \mapsto 2^S$ defines the set of available samples at time-point t for the fluent stream f:

$$available(f, t) \stackrel{\text{def}}{=} \{s \in f \mid atime(s) \leq t\}$$ □

The last available sample in a fluent stream is the sample with the greatest available time. In order to be able to reason about the past state of the system we also need to know which sample *was* the last available one at a given time-point t. We therefore introduce the function $last\_available(f, t)$.

Definition 4.3.8 (Last available sample) The function $last\_available(f, t)$: $F \times T \mapsto S$ defines the last available sample at time-point t for the fluent stream f:

$$last\_available(f, t) \stackrel{\text{def}}{=} \begin{cases} \texttt{no\_sample} & \text{if } available(f, t) = \emptyset \\ \underset{s \in available(f,t)}{\arg\max} \; atime(s) & \text{otherwise.} \end{cases}$$

□

Note that the valid time is independent of the available time, so even if the samples are totally ordered by available time they do not have to be ordered by valid time.

Example 4.3.6 (Fluent stream cont.) Continuing Example 4.3.5 the last available sample for f_1 at time-point 2 is $\langle 1, 1, v_1 \rangle$ and it is $\langle 3, 2, v_2 \rangle$ at time-point 3. The last

available sample for f_2 at time-point 1 is `no_sample` since it has no sample with an available time less than or equal to 1. At time-point 3 the last available sample is $\langle 2, 1, v_4 \rangle$ and at time-point 4 it is $\langle 4, 1, v_5 \rangle$. □

We are often interested in querying a fluent stream f to find the value of the associated feature at a given (valid) time-point t. However, a fluent stream rarely contains a sample for every possible valid time. One possible approximation would be to query f for a sample with the highest valid time less than or equal to t, among those samples that are available at the time when the query is asked t_q. A knowledge process can then make the assumption that this value may persist until t or use filtering, interpolation, or similar techniques to find a better estimate of the true value at t.

We will define this query function in three steps. First, we define the function $valid\_before(f, t, t_q)$, which returns all samples in the fluent stream f that are valid before or at time t and that are available at the query time t_q.

Definition 4.3.9 (Valid before) The function $valid\_before(f, t, t_q)$: $F \times T \times T \mapsto 2^S$ defines the set of samples which are available at t_q and have a valid time less than or equal to t.

$$valid\_before(f, t, t_q) \stackrel{\text{def}}{=} \{s \in available(f, t_q) \mid vtime(s) \leq t\}$$ □

Second, we introduce the function $last\_valid\_before(f, t, t_q)$, which returns those samples in $valid\_before(f, t, t_q)$ that have the greatest valid time. Note that since valid times are not guaranteed to be unique, this function must return a set of samples rather than a unique sample.

Definition 4.3.10 (Last valid before) The function $last\_valid\_before(f, t, t_q)$: $F \times T \times T \mapsto 2^S$ defines the set of samples which have the highest valid time less than or equal to t among those samples that are available at t_q.

$$last\_valid\_before(f, t, t_q) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } valid\_before(f, t, t_q) = \emptyset \\ \{s \in valid\_before(f, t, t_q) & \\ \mid vtime(s) = & \text{otherwise.} \\ \quad \arg\max_{s' \in valid\_before(f,t,t_q)} vtime(s')\} & \end{cases}$$

□

Finally, we must select a unique sample among those returned by the function $last\_valid\_before(f, t, t_q)$. We therefore introduce $most\_recent\_at(f, t, t_q)$. If there is more than one candidate (all of which have the same valid time), the best choice is likely to be the one with the highest available time. The knowledge process generating the stream would not have sent multiple samples with identical valid times unless the later values were likely to provide approximations of higher quality.

Definition 4.3.11 (Most recent at) The value of the function $most\_recent\_at(f, t, t_q) : F \times T \times T \mapsto S$ is the sample with the most recent valid time less than or equal to t among the available samples at t_q.

$$most\_recent\_at(f, t, t_q) \stackrel{\text{def}}{=} last\_available(last\_valid\_before(f, t, t_q), t_q) \qquad \square$$

Given a sample s in a fluent stream f, one would sometimes like to retrieve the sample that arrived immediately before s at the receiving process, or determine that s was in fact the first sample in f. We therefore introduce the function $prev(f, s)$, which makes use of the fact that the temporal domain is associated with a total order and that samples in a fluent stream are guaranteed to have unique available times.

Definition 4.3.12 (Previous sample) The function $prev(f, s) : F \times S \mapsto S$ defines the sample previous to s in the fluent stream f:

$$prev(f, s) \stackrel{\text{def}}{=} \begin{cases} \texttt{no\_sample} & \text{if } s = \texttt{no\_sample} \\ & \text{or } \neg\exists s' \in f.\, atime(s') < atime(s) \\ \underset{s' \in f \wedge atime(s') < atime(s)}{\arg\max} \; atime(s') & \text{otherwise.} \end{cases}$$

$\square$

Example 4.3.7 (Fluent stream cont.) Continuing Example 4.3.5 the previous sample of $\langle 3, 2, v_2 \rangle$ in f_1 is $\langle 1, 1, v_1 \rangle$, whose previous sample is `no_sample`. $\square$

Another function which is required by later definitions is $availabletimes(f)$ which defines the set of time-points when some sample was made available in a fluent stream f. The function is easily extended to return the set of available times for a set of fluent streams.

Definition 4.3.13 (Available times) The function $availabletimes(f) : F \mapsto 2^T$ defines the set of available times of a fluent stream f:

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

The function $availabletimes(\{f_1, \ldots, f_n\}) : 2^F \mapsto 2^T$ gives the set of available times for a set of fluent streams:

$$availabletimes(\{f_1, \ldots, f_n\}) \stackrel{\text{def}}{=} availabletimes(f_1) \cup \ldots \cup availabletimes(f_n).$$

$\square$

Example 4.3.8 (Fluent stream cont.) Continuing Example 4.3.5 the set of available times for f_1 is $\{1, 3, 4\}$ and for f_2 it is $\{2, 4, 5\}$. The set of available times for $\{f_1, f_2\}$ is $\{1, 2, 3, 4, 5\}$. $\square$

4.3.3 Source

To model a primitive process a *source* is introduced. A source can for example make the output of an external data producer or sensor available as streams. It provides a stream-based interface to the external producer. A source is represented by a function from time-points to samples, where the function determines the output of the source at each time-point. To represent that a source does not produce any value at a particular time-point the constant `no_sample` can be used.

Definition 4.3.14 (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 a domain D is denoted by R_D. □

Example 4.3.9 (Source) Some example sources are those providing interfaces to the GPS, IMU, and cameras on a UAV. Another example source is a process which reads a GUI where a user can enter observations about the number of cars in a region. □

4.3.4 Computational Unit

To model a special form of refinement process that has one output and only considers the most recent sample in each input stream a *computational unit* is introduced. A computational unit, which may have an internal state, processes the input from one or more streams, sample by sample, to compute a new stream. A computational unit with $n > 0$ input streams is associated with a partial function which takes 1 time-point, n samples, and 1 value representing the internal state as input and computes a pair consisting of a sample and a new internal state as output. The number n is the *arity* of the computational unit and $n + 2$ is the arity of the corresponding function. The initial internal state is `no_value`. The limitations on the knowledge processes that can be defined using computational units might seem restrictive, but for the applications developed so far it has matched the type of stream processing required. For example, it is possible to create a computational unit which has more than one output by computing samples with complex values.

Definition 4.3.15 (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 $T \times S_D^n \times V_D \mapsto S_D \times V_D$ mapping a time-point, n samples, and a value representing the previous internal state to an output sample and a new internal state. The set of all possible computational units in a domain D is denoted by C_D. □

Example 4.3.10 (Computational unit) An addition function could be defined as a computational unit whose input is a time-point, two integer-valued samples, and an internal state. This computational unit would ignore the time-point and internal state, returning a new sample whose value is the sum of the values of the input samples together with a new (arbitrary) internal state. Other examples are Kalman filters and functions which compute qualitative spatial relations between objects given their positions. □

A computational unit processes its input streams incrementally. This leads to the question of when to apply the function associated with the computational unit. Applying the function requires one sample from each input stream, but there is no guarantee that all input streams provide new samples with identical available times. For example, one input stream could be sampled with a period of 100 ms, another with a period of 60 ms, and a third could send samples asynchronously. In this case, which combinations of samples should the function be applied to?

One possibility would be to wait until new samples have been produced for every input stream before applying the function. However, this could lead to discarding a large number of samples from some input streams. Therefore, we choose to apply the function at every time t where there is a new sample in *some* input stream.

The next question to be answered is which samples the function should be applied to. Given that the function is applied at time t, there must be a sample with available time t in at least one input stream, but not necessarily in all. For those streams that do not provide a sample with an available time t, we can identify two options: Either we use the constant `no_sample` to represent that there is no sample, or we use the most recent sample in the input stream.

Since it is common for a computational unit to use the current value of each input the second option is chosen. Since the time-point t is provided as input to the computational unit function, it is easy for it to filter out all those samples which have an available time different from t and get the same result as if the first option had been chosen. The function *join* is introduced to provide a mathematical definition of the sequence of inputs to a computational unit function.

Definition 4.3.16 (Join) 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 which is the result of joining a sequence of fluent streams $f_1, \ldots, f_n$:

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

□

Example 4.3.11 (Fluent stream cont.) Continuing Example 4.3.5, the result of joining the fluent streams f_1 and f_2 is the stream $[\langle 1, 1, [\langle 1, 1, v_1\rangle, \texttt{no\_sample}]\rangle, \langle 2, 2, [\langle 1, 1, v_1\rangle, \langle 2, 1, v_4\rangle]\rangle, \langle 3, 3, [\langle 3, 2, v_2\rangle, \langle 2, 1, v_4\rangle]\rangle, \langle 4, 4, [\langle 4, 5, v_3\rangle, \langle 4, 1, v_5\rangle]\rangle, \langle 5, 5, [\langle 4, 5, v_3\rangle, \langle 5, 1, v_6\rangle]\rangle]$. A visualization of the result is shown in Figure 4.3. □

When a computational unit takes several inputs it is not always obvious what the valid time of the computed value should be. If the inputs to a computational unit have identical valid times then most likely the valid time of the output should be the same. For example, if we have a computational unit that calculates the distance between two positions and the two inputs have the same valid times then the distance sample would also have the same valid time. However, if the positions

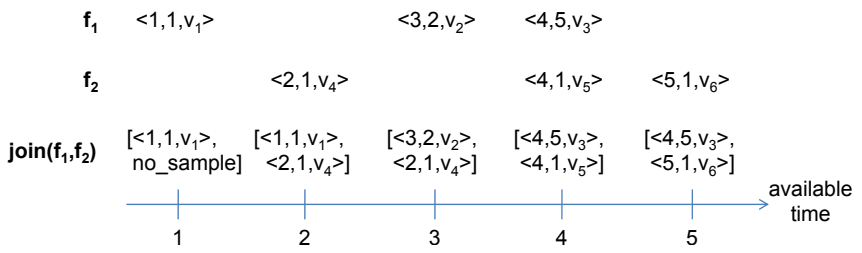


Figure 4.3: An example of two fluent streams and the result of joining them.

have different valid times, then what should the valid time of the result be? One option is to not compute any value unless the valid times are the same, or to estimate the values of each input at the same valid time. Since this is inherently dependent on the computation being made the choice has to be part of the computational unit function.

4.4 Syntax

Stream-based knowledge processing middleware defines four types of processes. Two of them, primitive and refinement processes, describe the processing made in a knowledge processing application in the form of a static network of knowledge processes and streams. The other two, mediation and configuration processes, describe how to change the network of knowledge processes over time. This section describes the syntax of the language KPL which is used to specify a static network of primitive and refinement processes, leaving mediation and configuration processes for future work.

A DyKnow application consists of sources, computational units, stream generators, and streams. Sources provide an interface to external information producers and make their output available as streams. This provides input to the application. Each source is declared by a *source declaration* "**source** *type name*", where *name* is a source symbol.

The set of computational units represents computations that the system can perform on streams. Compared to a source a computational unit is parameterized, which allows the application of the same computation to different input streams. For example, a parameterized speed estimation computational unit can be applied to any stream of position estimates. Each computational unit is declared with a *computational unit declaration* "**compunit** *type name(argument types)*", where *name* is a computational unit symbol. This computational unit can then be instantiated with different input streams as many times as necessary.

Every source and every instantiation of a computational unit corresponds to a knowledge process which has a stream generator. The streams created by a stream generator often represent approximations of an attribute of an object or a relation between objects. We therefore introduce structured names, *labels*, which consist of a feature symbol and zero or more object symbols. Given a source *s* it is possible

to declare a stream generator with "**strmgen** *label* = s", and given a computational unit *c* it is possible to instantiate it and declare a stream generator with "**strmgen** *label* = c(*input stream terms*)". These declarations are called *fluent stream generator declarations*.

From a stream generator it is possible to create many streams, where each stream is associated with its own policy. Every such stream is specified by a *stream term*, which has the form "*label* **with** *policy*". A stream created by a stream generator can be used either as an input to a computational unit or, in case not all processes are fully integrated into DyKnow, as an output of the application. In the first case, the stream term would be used in a stream generator declaration. In the second case, the stream term would be used in a *stream declaration* of the form "**stream** *name* = *stream term*", where *name* is a stream symbol.

Example 4.4.1 (KPL Example) To illustrate the use of KPL a small knowledge processing application where the speed of two cars are estimated will be used throughout the rest of this chapter. There are two sources, pos_car1 and pos_car2, providing information about the position of the two different cars and a computational unit SpeedEst which can estimate the speed of a car from a stream of position estimations. An overview of the processes, stream generators, and streams related to the first car is shown in Figure 4.4.

To specify the first source pos_car1 and a fluent stream generator pos[car1] from the source, the following declarations could be used:

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

The first statement declares that the symbol pos_car1 denotes a source which provides data of the type `pos`. The second statement declares a fluent stream generator pos[car1] which belongs to a primitive process instantiated from the source pos_car1.

To specify the computational unit SpeedEst and a fluent stream generator speed[car1] defined as SpeedEst applied to a fluent stream generated from pos[car1] by sampling it every 200 time units, the following declaration could be used:

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

The first statement declares that SpeedEst is a computational unit of arity 1 that takes samples of the sort `pos` and computes samples of the sort `speed`. The second statement declares a fluent stream generator speed[car1]. It states that the fluent stream generator is part of a refinement process which is created by instantiating the computational unit pos[car1]. The input to the refinement process is a single fluent stream, generated by the stream generator denoted by pos[car1] using the fluent stream policy "**sample every** 200".

To specify that an output of the application is a stream speed_car1 generated by the fluent stream generator speed[car1] between time-point 300 and time-point 400 the following declaration could be used:

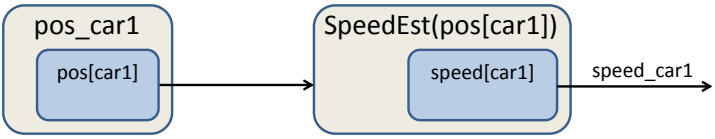


Figure 4.4: The processes, stream generators, and streams related to car1 specified in Example 4.4.1.

stream speed_car1 = speed[car1] **with from** 300 **to** 400

The processing of the information related to the second car is done in the same fashion as for the first car. A source declaration and a fluent stream generator declaration are used to specify a primitive process. Its stream generator makes the position observations available to the application. Another fluent stream generator declaration is used to specify a refinement process estimating the speed of the second car. To give a second example of a fluent stream policy, the computational unit is parameterized with a fluent stream whose delay is at most 100 time units and where the samples are made available ordered by their valid times. Finally, a fluent stream declaration is used to specify a stream containing the speed estimations as an output of the application.

source `pos` pos_car2
strmgen pos[car2] = pos_car2
strmgen speed[car2] = SpeedEst(pos[car2] **with max delay** 100,
monotone order)
stream speed_car2 = speed[car2] □

4.4.1 Vocabulary

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

Definition 4.4.1 (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 each associated with an arity > 0 defining how many arguments it takes,
- $\mathcal{N}$ 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 defining how many arguments it takes,

- $\mathcal{T}$ is a set of time-point symbols, and
- $\mathcal{V}$ 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. □

Definition 4.4.2 (KPL Symbols) The KPL symbols are comma, equals, left and right parenthesis, left and right bracket, and the set of 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**}. □

Example 4.4.2 (Signature) The signature of the knowledge processing application in Example 4.4.1 consists of the following domain dependent symbols: The two source symbols pos_car1 and pos_car2, and the computational unit symbol SpeedEst of arity 1. The object symbols car1 and car2 which represent the two cars and the feature symbols pos and speed both of arity 1 which represent the parameterized position and speed features. The stream symbols speed_car1 and speed_car2 which represent the two output streams. The set of time-point symbols which corresponds to the set of standard symbols for the integers in $\mathbb{Z}^+$. Finally, the value sort symbols `pos`, `speed`, `time`, and `object` which represent the sets of positions, speeds, time-points, and object constants respectively.

In formal notation, the signature $\sigma = \langle$ {car1, car2}, {pos/1, speed/1}, {speed_car1, speed_car2}, {pos_car1, pos_car2}, {SpeedEst/1}, $\mathbb{Z}^+$, {`pos`, `speed`, `time`, `object`} $\rangle$. □

4.4.2 KPL Specification

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 primitive process prototype (Section 4.4.3). A computational unit declaration, labeled **compunit**, declares a parameterized computational unit corresponding to a refinement process prototype (Section 4.4.3). A fluent stream generator declaration, labeled **strmgen**, declares a specific fluent stream generator created from either a source or a computational unit corresponding to a knowledge process (Section 4.4.4). A fluent stream declaration, labeled **stream**, declares a fluent stream which is generated by the application as an output (Section 4.4.5). These statements will be defined in detail in the following sections.

Definition 4.4.3 (KPL Specification) A KPL *specification* for a signature σ in KPL is a set of source declarations, computational unit declarations, fluent stream generator declarations, and fluent stream declarations for σ. □

The formal grammar for a KPL specification is defined as follows:

`KPL_SPEC ::= ( SOURCE_DECL`

```
                                | COMP_UNIT_DECL
                                | FLUENT_STREAM_GEN_DECL
                                | FLUENT_STREAM_DECL )+
                SOURCE_DECL ::= source SORT_SYMBOL SOURCE_SYMBOL
              COMP_UNIT_DECL ::= compunit SORT_SYMBOL
                                 COMP_UNIT_SYMBOL
                                 '(' SORT_SYMBOL
                                 ( ',' SORT_SYMBOL )* ')'
    FLUENT_STREAM_GEN_DECL ::= strmgen LABEL '='
                             (   SOURCE_SYMBOL
                             |   COMP_UNIT_SYMBOL
                                 '(' FLUENT_STREAM_TERM
                                 ( ',' FLUENT_STREAM_TERM )* ')'
                             )
        FLUENT_STREAM_DECL ::= stream STREAM_SYMBOL '='
                                 FLUENT_STREAM_TERM
                     LABEL ::= FEATURE_SYMBOL
                                 ( '[' OBJECT_SYMBOL
                                  ( ',' OBJECT_SYMBOL )* ']' )?
       FLUENT_STREAM_TERM ::= LABEL
                                 ( with FLUENT_STREAM_POLICY )?
     FLUENT_STREAM_POLICY ::= STREAM_CONSTRAINT
                                 ( ',' STREAM_CONSTRAINT )*
        STREAM_CONSTRAINT ::= APPROXIMATION_CONSTRAINT
                            | CHANGE_CONSTRAINT
                            | DELAY_CONSTRAINT
                            | DURATION_CONSTRAINT
                            | ORDER_CONSTRAINT
APPROXIMATION_CONSTRAINT ::= no approximation
                            | use most recent
        CHANGE_CONSTRAINT ::= any update
                            | any change
                            | sample every TIME_SYMBOL
         DELAY_CONSTRAINT ::= max delay ( TIME_SYMBOL | oo )
      DURATION_CONSTRAINT ::= from TIME_SYMBOL
                                 ( to ( TIME_SYMBOL | oo ) )?
                            | to ( TIME_SYMBOL | oo )
         ORDER_CONSTRAINT ::= any order
                            | monotone order
                            | strict order
```

4.4.3 Knowledge Process Declaration

A knowledge process declaration specifies a *class* of processes, closely related to the concept of a *knowledge process prototype* in the DyKnow implementation. This

class can be instantiated to create concrete knowledge processes, as shown in the following section.

Source Declaration

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 for any process interested in its output.

Definition 4.4.4 (Source declaration) A *source declaration* for a KPL signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form **source** v s, where v is a value sort symbol in $\mathcal{V}$ and s is a source symbol in $\mathcal{S}$. □

Example 4.4.3 (Source declaration) The source pos_car1 from Example 4.4.1 provides samples of the sort `pos` representing estimations of the position of car1. This is specified by a source declaration **source** `pos` pos_car1. □

Computational Unit Declaration

A computational unit declaration specifies a class of *refinement processes*. The specification includes the value sorts of its inputs and outputs, which implicitly defines the arity of the computational unit. A computational unit can be instantiated multiple times with different inputs and the stream generator for each instantiation can provide an arbitrary number of output streams for any process interested in its output.

Definition 4.4.5 (Computational unit declaration) A *computational unit declaration* for a KPL 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 $v_0, \ldots, v_n$ are value sort symbols in $\mathcal{V}$ and c is a computational unit symbol in $\mathcal{C}$. □

Example 4.4.4 (Computational unit declaration) The SpeedEst computational unit with arity 1 from Example 4.4.1 takes samples of the sort `pos` representing estimations of the position of an object as input and computes a sample of the sort `speed` representing an estimation of the speed of the same object. This is specified by the computational unit declaration **compunit** `speed` SpeedEst(`pos`). □

4.4.4 Fluent Stream Generator Declaration

Each fluent stream generator in a DyKnow application must be explicitly declared in KPL.

In the current version of KPL, each knowledge process is assumed to have a single output. Therefore, knowledge processes do not need to be explicitly instantiated through a separate process declaration. Instead a fluent stream generator

declaration can be used to specify both a generator and the knowledge process instance with which it is associated. The name given to the fluent stream generator is a *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.

Definition 4.4.6 (Label term) A *label term* for a KPL 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 $n \geq 0$, f is a feature symbol in $\mathcal{F}$ with arity n, and $o_1, \ldots, o_n$ are object symbols in $\mathcal{O}$. If $n = 0$ then the brackets are optional. □

Example 4.4.5 (Label term) Using the object and feature symbols from Example 4.4.1, two example labels are pos[car1] and speed[car2]. These labels could represent that fluent streams generated by the denoted fluent stream generators are approximations of the position of car 1 and the speed of car 2 respectively. □

A fluent stream generator can be declared in two different ways in KPL. The first way is to declare a fluent stream generator from a source, which corresponds to a primitive process. The other way corresponds to declaring a refinement process by instantiating a computational unit with fluent stream terms.

Definition 4.4.7 (Fluent stream generator declaration) A *fluent stream generator declaration* for a KPL 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. □

Example 4.4.6 (Primitive process) Continuing Example 4.4.1, a fluent stream generator pos[car1] of a primitive process instantiated from the source pos_car1 can be specified as:

strmgen pos[car1] = pos_car1 □

Example 4.4.7 (Refinement process) Continuing Example 4.4.6, the following statement can be used to specify a fluent stream generator speed[car1] belonging to a refinement process instantiated from the SpeedEst computational unit with an input fluent stream sampled from pos[car1] every 200 time units:

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

4.4.5 Fluent Stream Declaration

A fluent stream and its properties are specified by a fluent stream term. Each fluent stream term specifies the label of the fluent stream generator which provides the stream with content and a fluent stream policy specifying the constraints on the stream. A fluent stream can either be an input to a computational unit or an output of a knowledge processing application. A fluent stream declaration is used to specify a fluent stream generated as output by an application.

Fluent Stream Policy Specification

A fluent stream policy is a set of fluent stream constraints which specifies the properties of a fluent stream. A constraint can for example specify a regular sampling period or the duration of a fluent stream. There are currently five different constraint types: Approximation constraints, change constraints, delay constraints, duration constraints, and order constraints. Each of these is described in detail below.

Definition 4.4.8 (Fluent stream policy specification) A *fluent stream policy specification* for a KPL signature σ has the form $c_1, \ldots, c_n$, where $n \geq 0$ 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. □

Change Constraint Specification

A change constraint restricts the relation between consecutive samples in a fluent stream. Each sample except the first one must correspond to a change relative to the previous sample according to the change constraint. This restricts what samples may be added to a fluent stream. The change constraints defined in KPL are:

- Any update: Either the value or one of the time stamps of a sample must be different compared to the previous sample. This is trivially true since the available times must be different.
- Any change: Either the value or the valid time of a sample must be different compared to the previous sample.
- Sample every t time units: The difference in valid time between each pair of consecutive samples should be equal to the sample period t. This change constraint is often referred to as a *sample change constraint*.

Definition 4.4.9 (Change constraint specification) A *change constraint specification* for a KPL 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 $\mathcal{T}$. □

Example 4.4.8 (Change constraint example) A typical example is an application that needs a fluent stream where samples are added with a certain sample period, for example every 100 time units. This would be specified using a sample change constraint **sample every** 100. □

If no change constraint is specified then it is the same as specifying the any update constraint, **any update**.

Delay Constraint Specification

A delay constraint restricts the difference between the valid time and the available time of each sample in a fluent stream. This specifies the maximum delay that can be accepted for a fluent stream. The delay of a sample does not have to be caused by processing and communication, it could also be intentional to satisfy other constraints on a fluent stream. For example, it could be caused by waiting for delayed or missing samples.

Definition 4.4.10 (Delay constraint specification) A *delay constraint specification* for a KPL 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 $\mathcal{T}$. □

Example 4.4.9 (Delay constraint example) A typical example of a delay constraint is to specify that the maximum acceptable delay is 100 time units. This would be specified by a delay constraint **max delay** 100. □

Not specifying any delay constraint is equivalent to specifying an infinite delay, **max delay oo**.

Duration Constraint Specification

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

Definition 4.4.11 (Duration constraint specification) A *duration constraint specification* for a KPL 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 $\mathcal{T}$. □

Example 4.4.10 (Duration constraint example) A typical example of a duration constraint is to specify that the interesting valid times of a fluent stream are those between time-point 300 and time-point 400. This would be specified with a duration constraint **from** 300 **to** 400. □

If the start time or the end time is left out then there is no restriction on it. If 0 is the first time-point, then **to** t is equivalent to **from** 0 **to** t and **from** t is equivalent to **from** t **to oo**. If no duration constraint is specified then it is the same as specifying an infinite duration.

Order Constraint Specification

The definition of a fluent stream requires that the samples are ordered by available time, but the valid times of the samples may have any order. To specify a restriction on the relation between the valid times of consecutive samples an order constraint is introduced. The possible order constraints are:

- Any order: No restriction on the order of samples according to valid times.
- Monotone order: Each sample must have a valid time not less than the valid time of the sample before it.
- Strict monotone order: Each sample must have a valid time greater than the valid time of the sample before it.

Definition 4.4.12 (Order constraint specification) An *order constraint specification* for a KPL signature σ has either the form **any order**, the form **monotone order**, or the form **strict order**. □

Example 4.4.11 (Order constraint example) An order constraint which specifies that the samples in a fluent stream should be ordered by their valid times and that the fluent stream is allowed to contain more than one sample with the same valid time is written **monotone order**. □

If a sample change constraint is specified for a fluent stream then it implies a strict monotone order constraint. If no order constraint is specified then it is the same as specifying an any order constraint, **any order**.

Approximation Constraint Specification

An approximation constraint restricts how a fluent stream may be extended with new samples in order to satisfy its policy. The idea is that if the fluent stream generated by a knowledge process does not contain the appropriate samples to satisfy a policy, then a fluent stream generator could approximate the missing samples based on the available samples. The approximation constraints defined in KPL are:

- No approximation: No approximated samples are allowed to be added to a fluent stream.
- Use most recent: Suppose other parts of a fluent stream policy require the existence of a sample with a particular valid time t_v to be made available at or before t_q. "Use most recent" means that if no such sample arrives at the fluent stream generator, it is allowed to generate an approximate sample having the same value as the sample with the highest valid time less than or equal to t_v among the available samples at t_q.

In order for the stream generator to be able to determine at what valid time a sample must be produced, this constraint can only be used in conjunction with a complete duration constraint specification of the form **from** t_f **to** t_t and a change

constraint specification of the form **sample every** t_s. In order 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 specification of the form **max delay** t_d.

Definition 4.4.13 (Approximation constraint specification) An *approximation constraint specification* for a KPL signature σ has either the form **no approximation** or the form **use most recent**. □

Example 4.4.12 (Approximation constraint example) A typical example is when a process needs a fluent stream containing samples with a certain sample period and it would like the fluent stream generator to use the most recent value in case a sample is missing. This would be specified by an approximation constraint **use most recent** combined with a sample change constraint and a delay constraint. □

If a policy does not contain an approximation constraint then it is the same as specifying the no approximation constraint, **no approximation**.

Fluent Stream Term

A fluent stream term specifies a single fluent stream. The term consists of a label denoting the fluent stream generator which generates the stream and a fluent stream policy specification which specifies the desired properties of the stream.

Definition 4.4.14 (Fluent stream term) A *fluent stream term* for a KPL signature σ has the form l **with** p, where l is a label term for σ and p is a fluent stream policy specification for σ. If the policy p is the empty string then the keyword **with** can be left out. □

Example 4.4.13 (Fluent stream term) A fluent stream generated by the fluent stream generator labeled pos[car1] having the property of being sampled every 200 time units could be specified by "pos[car1] **with sample every** 200". □

Fluent Stream Declaration

A fluent stream declaration specifies a fluent stream that is generated as an output of a knowledge processing application.

Definition 4.4.15 (Fluent stream declaration) A *fluent stream declaration* for a KPL 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. □

Example 4.4.14 (Fluent stream declaration) The fluent stream speed_car1 from Example 4.4.1 is generated from the fluent stream generator speed[car1] with a policy stating that the duration of the stream is between time-point 300 and time-point 400. This can be specified by the following fluent stream declaration:

stream speed_car1 = speed[car1] **with from** 300 **to** 400. □

4.5 Semantics

In the preceding sections, we have defined two important entities.

A knowledge processing domain specifies the set of objects, time-points, and simple values that are available in a particular knowledge processing application. This indirectly defines the possible complex values, the sources and computational units that could potentially be defined over these values, and the fluent streams that they could produce.

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*.

Definition 4.5.1 (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 $\langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$, where:

- I_O is a function mapping each object symbol in $\mathcal{O}$ to a distinct object in O,
- I_F is a function mapping each feature symbol with arity n in $\mathcal{F}$ to a function $O^n \mapsto F_D$ mapping each instantiated feature to a fluent stream in F_D,
- I_N is a function mapping each stream symbol in $\mathcal{N}$ to a fluent stream in F_D,
- I_S is a function mapping each source symbol in $\mathcal{S}$ to a function in R_D,
- I_T is a function mapping each time-point symbol in $\mathcal{T}$ to a distinct time-point in T, and
- I_V is a function mapping each value sort symbol in $\mathcal{V}$ to a set 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 4.5.2 (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 $D = \langle O, T, P \rangle$ be a domain. Then, a *knowledge process specification* K_C for σ and D 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 4.5.1 (Interpretation and knowledge process specification) 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 4.4.2 on page 48 to the domain $D = \langle O, T, P \rangle$ from Example 4.3.1 on page 37 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 and car2 to o_4.
- I_F maps pos and speed to unary functions. The unary function associated with pos maps o_3 (car1) to the fluent stream $f_1 = [\langle 250, 200, p_1 \rangle, \langle 325, 300, p_2 \rangle, \langle 430, 400, p_3 \rangle, \langle 505, 500, p_4 \rangle]$, o_4 (car2) to the fluent stream $f_2 = [\langle 150, 100, p_6 \rangle, \langle 250, 180, p_7 \rangle, \langle 480, 400, p_8 \rangle]$, and all other objects to the empty fluent stream. The unary function associated with speed maps o_3 to the fluent stream $f_3 = [\langle 510, 400, s_3 \rangle]$, o_4 to the fluent stream $f_4 = [\langle 300, 100, s_6 \rangle]$, and all other objects to the empty fluent stream.
- I_N maps speed_car1 to the fluent stream $f_5 = [\langle 345, 300, p_2 \rangle, \langle 460, 400, p_3 \rangle]$ and speed_car2 to the fluent stream f_4,
- I_S maps pos_car1 and pos_car2 to unary functions. The function associated with pos_car1 maps 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 all other time-points to `no_sample`. The function associated with pos_car2 maps 150 to $\langle 150, 100, p_6 \rangle$, 250 to $\langle 250, 180, p_7 \rangle$, 480 to $\langle 480, 400, p_8 \rangle$, and all other time-points 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 the same signature is K_C which maps SpeedEst to the computational unit c taking a position value p_i as input and computing the speed value s_i as output. □

4.5.1 Model

A KPL specification s is a set of source declarations, computational unit declarations, fluent stream generator declarations, and fluent stream declarations. To define whether an interpretation satisfies a KPL specification given a knowledge process specification the relation $\models$ is introduced. If an interpretation satisfies a KPL specification given a knowledge process specification then it is said to be a *model* of the specification.

Definition 4.5.3 (Model) Let σ be a signature, D be a domain, I be an interpretation of σ to D, K_C be a knowledge process specification for σ and D, and s be a KPL specification for σ. Then, I is a *model* of s given K_C, written $I, K_C \models s$, if and only if for every declaration $d \in s, I, K_C \models d$. □

4.5.2 Knowledge Process Declaration

A knowledge process declaration constrains the value domain of the samples produced by any instantiated knowledge process. A computational unit declaration also constrains the value domains of the fluent streams which can be used to instantiate refinement processes from the computational unit. To extract the values used by a set of samples the function *values* is introduced.

Definition 4.5.4 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$ be an interpretation for σ, K_C a knowledge process specification for σ, and d be a knowledge process 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 $T \times S_1 \times \ldots \times S_n \times V \mapsto S_0 \times V$

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

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

□

Definition 4.5.5 (Values) The function $values(s) : 2^S \mapsto 2^V$ defines the set of values used by a set of samples s.

$$values(s) \stackrel{\text{def}}{=} \{val(sa) \mid sa \in s \wedge sa \neq \texttt{no\_sample}\}$$ □

Example 4.5.2 The source declaration "**source** `pos` pos_car1" is satisfied by the interpretation I from Example 4.5.1 since the set of values of the function associated with pos_car1 is $\{p_1, p_2, p_3, p_4\}$ which is a subset of $I_V(\texttt{pos}) = \{p_1, \ldots, p_{10}\}$.
□

4.5.3 Fluent Stream Generator Declaration

The task of a fluent stream generator is to take the output of a source or a computational unit and provide a facility where other processes can ask for output streams adapted to given policies. In the declarative KPL language, we assume that there is a distinct adapted output stream for each occurrence of a fluent stream term, such as "pos[car1] **with sample every** 200". To improve modularity, the adaptation of the raw output of a knowledge process will be handled in the semantics of such fluent stream terms. The semantics of a fluent stream generator declaration therefore becomes quite simple, essentially describing how the unadapted input to the generator depends on the knowledge process which it is a part of.

Each fluent stream generator is associated with a label term consisting of a feature symbol and possibly a sequence of object symbols. For convenience we choose to let this label term denote a fluent stream corresponding to the input to the associated fluent stream generator. We introduce the following shorthand notation for evaluating a complete label term in an interpretation.

Definition 4.5.6 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$ be an interpretation for σ, and $f[o_1, \ldots, o_n]$ be a label term for σ. Then,

$$eval\_label(I, f[o_1, \ldots, o_n]) \stackrel{\text{def}}{=} I_F(f)(I_O(o_1), \ldots, I_O(o_n)) \qquad \square$$

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 4.5.7 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$ be an interpretation for σ, K_C be a knowledge process specification for σ, s be a source symbol for σ, and l be a label term for σ. Then,

$$I, K_C \models \textbf{strmgen}\ l = s \quad \text{iff} \quad eval\_label(I, l) = \{sa \mid \exists t. I_S(s)(t) = sa \wedge sa \neq \texttt{no\_sample}\}$$

$\square$

A computational unit calculates a new 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. To evaluate a fluent stream term in a given interpretation the function *eval_fsterm* is used. It will be defined in the next section. Since there might be several fluent streams which satisfies a fluent stream term *eval_fsterm* returns a set of fluent streams.

Definition 4.5.8 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V \rangle$ be an interpretation for σ, K_C be a knowledge process specification for σ, 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 $\overline{s}$ to denote a sequence of samples of appropriate length for the context in which it appears. Then,

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

$\square$

Example 4.5.3 (Fluent stream generator) The fluent stream generator declaration "**strmgen** pos[car1] = pos_car1" is satisfied by the interpretation I from Example 4.5.1, since $I_F(\mathsf{pos})(I_O(\mathsf{car1})) = f_1$ which contains the same samples as the function denoted by pos_car1. $\square$

4.5.4 Fluent Stream Declaration

A fluent stream term 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. We therefore take care to provide an interpretation of a fluent stream term as one from a *set* of possible streams, giving implementations some freedom in choosing how policies are applied while still ensuring that all constraints are met.

To define the semantics of a fluent stream term we need to introduce the substream relation $\sqsubseteq$. It is similar to, but not the same as the subset relation since it takes the available time into account. The intuition is that if a fluent stream f' is a substream of f then f' could be generated from f. A stream f' is a substream of a fluent stream f iff for every sample in f', f contains a sample with same value and valid time but an available time which is equal to or earlier than the valid time of the sample in f'. The reason for this definition is that a generated fluent stream might have different available times than the original fluent stream. However, it may never change the value or the valid time of a sample.

Definition 4.5.9 (Substream relation) A fluent stream f' is a substream of a fluent stream f, written $f' \sqsubseteq f$, iff $\forall \langle t'_a, t_v, v\rangle \in f' \; \exists t_a.\; [t_a \leq t'_a \wedge \langle t_a, t_v, v\rangle \in f]$. □

Example 4.5.4 (Substream relation) Using the fluent streams from the interpretation I in Example 4.5.1, the fluent stream $f_5 = [\langle 345, 300, p_2\rangle, \langle 460, 400, p_3\rangle]$ is a substream of the fluent stream $f_1 = [\langle 250, 200, p_1\rangle, \langle 325, 300, p_2\rangle, \langle 430, 400, p_3\rangle, \langle 505, 500, p_4\rangle]$ since each of the two samples in f_5 have corresponding samples in f_1 where only the available time differs and is earlier in f_1. □

A fluent stream term consists of a label term l and a (possibly empty) policy p. The interpretation of the label term, $eval\_label(I, l)$, is a fluent stream representing the unadapted output provided to the associated fluent stream generator. The interpretation of the fluent stream term, $eval\_fsterm(I, l$ **with** $p)$, 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. The intuition is that 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. We define maximal in terms of set inclusion. For a stream to be maximal it must not be a subset of another substream of the extended stream which satisfies the same policy p.

Definition 4.5.10 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V\rangle$ be an interpretation for σ, and l **with** p be a fluent stream term for σ. The interpretation of

the fluent stream term, *eval_fsterm*$(I, l$ **with** $p)$ is then defined as follows:

$$
\begin{aligned}
\mathit{eval\_fsterm}(I, l \textbf{ with } p) &\stackrel{\text{def}}{=} \{f \in \mathit{satisfying}(I, l \textbf{ with } p) \\
&\quad \mid \neg\exists f' \in \mathit{satisfying}(I, l \textbf{ with } p).f \subset f'\} \\
\mathit{satisfying}(I, l \textbf{ with } p) &\stackrel{\text{def}}{=} \{f \sqsubseteq \mathit{extend}(\mathit{eval\_label}(I, l), p) \mid I, f \models p\} \\
\mathit{extend}(f, p) &\stackrel{\text{def}}{=} \begin{cases}
\{\langle t_a, t_v, v\rangle \quad \mid & \text{if } \textbf{use most recent} \in p \\
\exists n \geq 0.t_v = b + sn \wedge t_v \leq e & \wedge\exists s.\textbf{sample every } s \in p \\
\wedge t_a = t_v + d & \wedge\exists b, e.\textbf{from } b \textbf{ to } e \in p \\
\wedge v = \mathit{most\_recent\_at}(f, t_v, t_a)\} & \wedge\exists d.\textbf{max delay } d \in p \\
f & \text{otherwise.}
\end{cases}
\end{aligned}
$$

□

The policy associated with a fluent stream term is used to filter out all streams which are not valid according to the policy. This is used to constrain the set of streams that can be generated by a specified knowledge processing application.

Definition 4.5.11 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V\rangle$ be an interpretation for σ, f be a fluent stream, and p be a fluent stream policy specification for σ. Then, $I, f \models p$ according to:

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

□

Example 4.5.5 (Fluent stream policy) The fluent stream policy specification "**sample every** 100" is satisfied by the fluent stream f_1 in the interpretation I from Example 4.5.1, since the difference between each pair of valid times is exactly 100 time units. The same fluent stream does not satisfy the policy specification "**max delay** 40" since the first sample has a delay of 50 time units. □

Example 4.5.6 (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 4.5.1, the fluent stream f is one of the possible streams that can be generated from the fluent stream term "pos[car1] **with sample every** 200". First of all f is a substream of the stream f_1 denoted by the label l as shown in Example 4.5.4. Second, f satisfies the policy p since the difference between each pair of valid times is exactly 200 time units. Finally, it is not possible to add any more of the samples from f_1 to f without violating the policy p. □

Finally, we define the semantics of a stream declaration which is quite trivial given the previous definitions.

Definition 4.5.12 Let σ be a KPL signature, $I = \langle I_O, I_F, I_N, I_S, I_T, I_V\rangle$ be an interpretation for σ, K_C be a knowledge process specification for σ, n be a stream symbol for σ, l be a label term for σ, and p be a fluent stream policy specification. Then,

$$I, K_C \models \textbf{stream}\ n = l\ \textbf{with}\ p \quad \text{iff} \quad I_N(n) = eval\_fsterm(I, l\ \textbf{with}\ p) \qquad \square$$

4.6 Summary

This chapter has described a concrete stream-based knowledge processing middleware framework called DyKnow. DyKnow defines two types of entities, objects and features. Since the value over time of a feature can not be completely known it is approximated by a fluent stream.

It is important to realize that there is not a single best approximation which can be used in all situations. Rather, what is an appropriate approximation will depend on the current task. When executing some tasks it is more important to have the most current information, even though it might be more uncertain and might contain occasional errors. This is usually the case for tasks controlling a piece of equipment by continuously making small corrections. Even if a correction is wrong once in a while it does not usually have any severe consequences. In other tasks, for example those involving collecting information which is to be used at a later date, it is more important that the information is as accurate as possible. In this case it is better to use potentially computationally expensive algorithms which correlate the value with other observations in order to estimate the best possible value. Some applications might even have to switch between tasks having different characteristics during execution depending on the current situation. Therefore it is important to be able to create many parallel approximations, to configure these approximations, and to switch between the approximations during execution.

To describe knowledge processing applications creating approximations in the form of fluent streams a formal language was introduced. The DyKnow knowledge processing language KPL is used to write declarative specifications of knowledge processing applications. The domain, the syntax, and the semantics of KPL was formally defined and exemplified.

Chapter 5

A DyKnow CORBA Middleware Service

5.1 Introduction

In this chapter we will describe the DyKnow CORBA middleware service, which supports the implementation of distributed knowledge processing applications according to KPL specifications.

As shown in the previous chapter, a knowledge processing application in DyKnow conceptually consists of a set of knowledge processes connected by streams satisfying policies, where each knowledge process is an instantiated source or computational unit (Figure 5.1). The knowledge processes provide stream generators which create streams. The KPL language, also introduced in the previous chapter, can be used to provide a declarative specification of a DyKnow application. The DyKnow service, in turn, takes a set of KPL declarations and sets up the required processing and communication infrastructure to allow knowledge processes

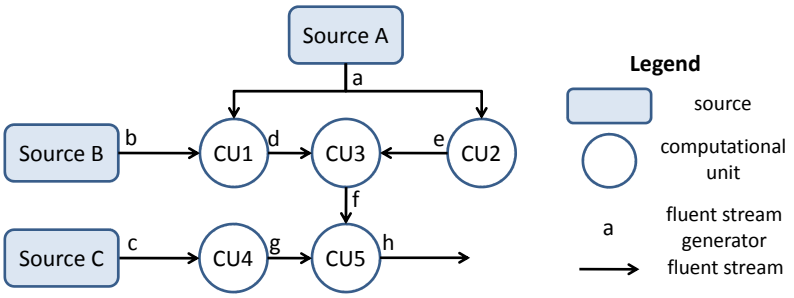


Figure 5.1: A conceptual view of a DyKnow knowledge processing application consisting of a set of sources and computational units consuming and producing fluent streams.

Figure 5.2: A conceptual overview of the DyKnow middleware service.

to work according to specification in a distributed system.

The DyKnow middleware service can take a complete KPL application specification as its input, resulting in a DyKnow application where the set of knowledge processes and the set of streams connecting them are static. It is also possible to specify an initial set of KPL specifications and then incrementally add new declarations.

To support distributed real-time and embedded systems, such as autonomous unmanned aerial vehicles, the DyKnow middleware service is implemented as a service in the Common Object Request Broker Architecture (CORBA) (Object Management Group, 2005, 2008). CORBA is an object-centric middleware where object-oriented applications can easily be developed disregarding the fact that objects can be implemented in any language and hosted on any computer in the network. As will be seen in the following section, another benefit of using CORBA is that we can build upon existing CORBA services, for example the naming service and the real-time notification service.

5.2 Overview

The DyKnow service supports the generation of streams from a set of knowledge processes, where the knowledge processes are not part of the DyKnow service itself. A process which interacts with a knowledge process through the DyKnow service is called a *client* (Figure 5.2). Since many knowledge processes subscribe to the output of other processes a knowledge process is often also a client.

A client can control a knowledge processing application by creating and destroying knowledge processes and accessing the output of existing knowledge processes through their stream generators. Creating and destroying stream generators corresponds to adding and removing stream generator declarations from an application specification.

Sources and parameterized computational units can be registered with the DyKnow service as *knowledge process prototypes*. Each registration corresponds to a source or computational unit declaration in KPL. A knowledge process prototype can then be instantiated to create knowledge processes. When a knowledge process is created it registers its stream generators with the service. It is also possible for already existing knowledge processes to register their stream generators directly with the middleware service without having to be instantiated. Each registered stream generator corresponds to a fluent stream generator declaration in KPL.

Conceptually, a stream is created by a stream generator which is part of a

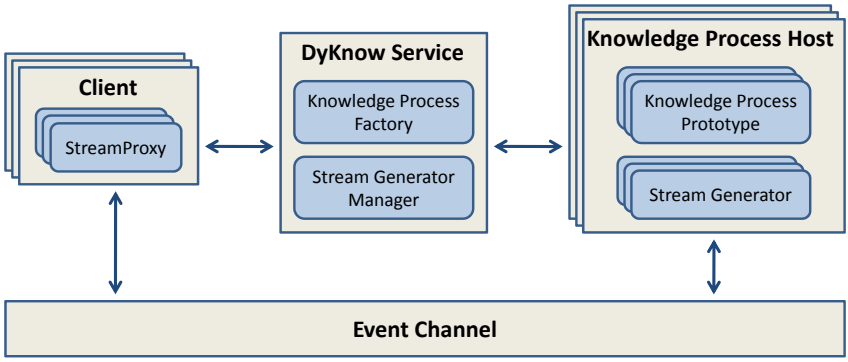


Figure 5.3: A high level overview of the interfaces and components of the DyKnow middleware service and with whom they interact.

knowledge process. The stream generator is responsible for making sure that the policy associated with the stream is satisfied, including approximating values if necessary. In a practical implementation special care has to be taken since values might be lost during communication over the network and must be approximated on the receiving side. It is also the case that the available time has to be assigned on the client side, when the information has been delivered in the potentially distributed system. This means that at least part of the policy must be implemented on the client side. We have therefore chosen to implement a *stream proxy* which is used on the client side and which ensures that the received stream satisfies the associated policy. Using a stream proxy makes it possible for a stream generator to broadcast samples over a CORBA *event channel* to distribute information to the clients. This decouples clients from stream generators.

In practice, it is unrealistic that each knowledge process is realized as a separate operating system process. DyKnow therefore supports the possibility for an operating system process to implement many knowledge processes. Such a process is called a *knowledge process host*.

A CORBA middleware service is defined by a set of interfaces. Some of these interfaces describe components of the service itself while the rest are interfaces to client code. To realize these functionalities the DyKnow service defines five interfaces: Stream Generator, Stream Generator Manager, Knowledge Process Factory, Knowledge Process Prototype, and Stream Proxy. Of these the service implements the Stream Generator Manager, the Knowledge Process Factory, and the Stream Proxy interfaces. The other two are used by application programmers to implement knowledge processes which can be used with the service. An overview of the different interfaces and components is shown in Figure 5.3.

5.2.1 DyKnow Service Dependencies

The DyKnow middleware service is dependent on four other services: A naming service, an event notification service, a time service, and an alarm service. The naming and the event notification services are provided by the TAO/ACE CORBA (Object Computing, Inc., 2003) implementation we use while the other two are designed and implemented by us.

Naming Service

Each CORBA object has a unique identity in the form of an interoperable object reference (IOR), which acts as a pointer to the object. One issue in an object-oriented middleware is how a client gets a pointer to a specific object. If the object is created by the client then it is not a problem, but in a distributed system the object may be created by anyone, anywhere. For example, a client might want to access a sensor, which is associated with an object providing an interface to it created by the program controlling the sensor. A common solution to this problem is to use a naming service which associates a name with an object IOR. This allows a client that knows the name of an object to look up the pointer to the object in the naming service. If no object exists with the desired name, then it either has to wait for the object to be created by someone else or create it itself. A name is usually a string, but more elaborate structures are also possible.

Event Notification Service

An event notification service implements an event channel where producers can publish data in the form of events through a consumer proxy interface and consumers subscribe to data through a supplier proxy interface. The events can be of any CORBA data type. An event channel supports many-to-many communication since there can be many producers and many consumers of events. Depending on the implementation the event channel may support different types of filtering and quality of service guarantees. DyKnow uses the event channel to deliver samples to subscribers. By using the filtering functionality the content of several streams can be distributed through a single event channel.

Time Service

A time service provides a global clock that keeps track of the current time. The time service is used to time-stamp samples. Since the time-stamps come from the same global clock they can be compared, which is essential when synchronizing streams as described in Section 7.8.

Alarm Service

In order to provide a stream generator with regular timeouts, which are needed to implement support for sample change constraints, an alarm service is used. An

alarm service supports the creation of timers that go off either at a specific time-point or with a particular timeout interval. The timeout should be synchronized with the time service as accurately as possible. This is important for the implementation of sampled streams.

5.3 Knowledge Process Host

Each knowledge process must be hosted and executed by some operating system process in the distributed system. This process is called the *host* of the knowledge process. From the point of view of the DyKnow service, it does not matter where a knowledge process is hosted as long as DyKnow can access the stream generators of the process. From an execution point of view it does make a difference since the delay before execution and the time to execute the knowledge process depends on the host. Contributing factors are for example the load on the host machine and the number of concurrent knowledge processes hosted. The communication delays are also influenced depending on whether the communicating knowledge processes are in the same host, in different hosts on the same machine, or in different hosts on different machines.

To make the hosted knowledge processes available to the DyKnow service each of them has to register its stream generators in the Stream Generator Manager. When a stream generator is registered then it is possible for clients to access it through the DyKnow service.

A knowledge process host can also support the creation of new knowledge processes by making knowledge process prototypes available. A prototype is made available by registering it in the Knowledge Process Factory.

A knowledge process host has the opportunity to decide when each of its knowledge processes should be executed and thus implement support for making a trade-off among its knowledge processes. This can be used to give some knowledge processes a higher priority and execute them before any process with a lower priority. It could also be used to support optimizations of the execution such as batch and train processing of samples. Batch processing is when the host waits for a batch of input samples for a knowledge process to be available before processing them all. Train processing is when a number of knowledge processes are connected in a chain and the chain is seen as an atomic operation where each input sample is processed from start to end of the chain before executing the next scheduled action.

5.3.1 Knowledge Process Prototype

Each knowledge process prototype provided by a host should implement the following interface.

- `create_instance`(*lbl*, *input_streams*, *output_policy*): Create an instance of the prototype by instantiating the prototype with *input_streams* which

is a sequence of label-policy pairs specifying the input streams. The *output_policy* parameter can be used to specify properties of the information provided by this knowledge process to its stream generator, such as a fundamental sample period for the entire process. This output can then be adapted further by the stream generator for every generated stream as described previously. The stream generator of the new knowledge process should be associated with the label *lbl* in the Stream Generator Manager. The method will return the stream generator of the new knowledge process.

If the knowledge process does not support the output policy then a PolicyNotSupported exception is thrown. If the number of input streams does not match the input arity of the prototype then a WrongNumberOfInputs exception is thrown.

Two types of knowledge process prototypes are sources and computational units. A source prototype corresponds to an external stream producer and an instantiated source provides an interface to a stream generated by this external producer. An instantiated source is an example of a primitive knowledge process. A computational unit prototype corresponds to a stream computation which takes at least one stream as input and computes a new stream as output. An instantiated computational unit is an example of a refinement knowledge process.

To make sources and computational units available to the DyKnow middleware service they have to be registered as knowledge process prototypes in the Knowledge Process Factory (see `add_prototype` in Section 5.4.1). They can then be instantiated with policies to create primitive and refinement processes at the host.

5.3.2 Stream Generator

A stream generator provides an interface to one of the outputs of a knowledge process. Conceptually, each output of a knowledge process is associated with a stream generator that can generate multiple output streams, each of which is adapted to satisfy a specific policy. In practice, as noted previously, some of the necessary processing must take place at the receiving end of the stream. In the current implementation, we have chosen to collect all processing at the client side in a *stream proxy*. The stream generator pushes all of its output on the CORBA event channel tagged with its own label, thereby making it available for any number of clients. When a stream generator is asked for a stream satisfying a given policy, a new stream proxy for that policy is generated on the client side. The stream proxy automatically connects to the event channel, subscribes to samples having the correct label, adapt the samples to its policy, and provides them to the client through a stream-based interface. From the client's point of view, the use of proxies and event channels is transparent.

A stream generator also provides an interface to query the knowledge process about the content of its output stream. If a stream generator provides a fluent stream then it is for example possible to ask for a sample in the stream with a particular valid time or to ask for all samples with a valid time in a certain range. To support

these queries and to support subscriptions starting in the past a stream generator may cache the stream created by the knowledge process.

A stream generator should implement the following interface.

- `get_size`(): Return the current number of elements in the cached stream.
- `get_latest`(): Return the latest element in the cached stream. This is the element with the highest available time among the elements generated so far. If no elements are cached then throw a ValueNotAvailable exception.
- `get_nth_latest`(*n*): Return the *n*:th latest element in the cached stream, where the 0:th latest element is the element with the highest available time.
- `get_slice`(*i*, *j*): Return the sequence of the *i*:th to the *j*:th elements of the cached stream. The elements are numbered starting at 0.
- `reset`(): Reset the stream generator, which means empty the cache and remove all computed elements.

A fluent stream generator should implement both the general stream generator interface and the following fluent stream specific interface.

- `get_latest_vtime`(): Return the sample with the highest valid time in the cached fluent stream. If there is more than one sample with the same valid time then return the one with the highest available time within this set. If there are no cached samples then throw a ValueNotAvailable exception.
- `get_closest_at_or_before`(*t*): Return the sample with the highest valid time which is less than or equal to *t* in the cached fluent stream. If there is no such sample then throw a ValueNotAvailable exception. If there is more than one sample with the same valid time then return the one with the highest available time within this set.
- `get_between`(*from*, *to*): Return a sequence of samples containing all samples with a valid time in the range [*from*, *to*] in the cached fluent stream. If there are no samples in the range then an empty sequence is returned.

One benefit of only specifying interfaces is that a knowledge process host can implement knowledge process prototypes and stream generators in the most appropriate way for its purpose. There are for example many ways of satisfying a declarative policy, especially when there is a need to estimate samples in the face of incomplete or uncertain information. DyKnow also provides standard implementations of stream generators.

5.4 The DyKnow Service

The DyKnow service provides the core knowledge processing functionality: The creation of new knowledge processes through the instantiation of knowledge process prototypes, the management of existing knowledge processes, and the creation

of streams from the output of knowledge processes. This section describes how these functionalities are realized.

5.4.1 The Knowledge Process Factory

The purpose of the Knowledge Process Factory is to allow clients to create knowledge processes by instantiating knowledge process prototypes provided by a knowledge process host. The factory has methods allowing a host to add and remove its knowledge process prototypes and methods allowing clients to create instances of these prototypes.

When a knowledge process host registers a prototype corresponding to a source it is the same as adding a source declaration **source** v s to the knowledge application specification. The declaration specifies the data type of the samples, v, and the name of the source, s. When a prototype corresponding to a computational unit is registered it is the same as adding a computational unit declaration **compunit** v_0 $c(v_1 \ldots, v_n)$ to the application specification. The declaration specifies the name of the computational unit, c, and the data types of its inputs, $v_1 \ldots v_n$, and output, v_0.

The knowledge process factory has two responsibilities. First, it should keep track of all the source and computational unit prototypes in a knowledge processing application. Second, it should create knowledge processes by instantiating these prototypes with policies. To perform these duties the knowledge process factory implements the following interface.

- `add_prototype`(*prototype*, *name*, *input_sorts*, *output_sort*): Add *prototype* as a knowledge process prototype associated with the string *name*. The arity and sorts of the input are specified by *input_sorts* which is a sequence of sorts. The sort of the output is specified by the sort *output_sort*. If *name* is already associated with a prototype then a PrototypeAlreadyExists exception is thrown.

- `remove_prototype`(*name*): Remove the prototype associated with the string *name*.

- `create_knowledge_process`(*lbl*, *name*, *input_streams*, *output_policy*): Create a knowledge process with a single stream generator by instantiating the prototype associated with the string *name*. The *output_policy* can be used to provide a hint to the knowledge process regarding the type of streams that will be generated from its stream generator, for example by providing a basic sample rate for a sensor process. The stream generator of the process is associated with the label *lbl*. It can further adapt the output from the knowledge process when asked for a stream satisfying a specific policy. If the label is already associated with a stream generator then a LabelAlreadyExists exception is thrown. If the knowledge process does not support the output policy then a PolicyNotSupported exception is thrown.

  The labels and policies used to create the input streams to the knowledge process are specified by *input_streams* which is a sequence of label-policy

pairs. If the number of input streams does not match the input arity of the prototype then a WrongNumberOfInputs exception is thrown. The method returns the stream generator of the newly created knowledge process.

- `destroy_knowledge_process`(*lbl*): Destroy the knowledge process of the stream generator associated with the label *lbl*. It is only possible to destroy knowledge processes which have been created by the factory. If there is no stream generator associated with the label *lbl* then a NoSuchLabel exception is thrown.

5.4.2 The Stream Generator Manager

To keep track of the stream generators in an application and their associated labels the Stream Generator Manager is used. The stream generator manager implements the following interface.

- `add_stream_generator`(*lbl*, *gen*): Add the stream generator *gen* and associate it with the label *lbl*. If a stream generator is already associated with the label *lbl* then a LabelAlreadyExists exception is thrown.

- `remove_stream_generator`(*lbl*): Remove the stream generator associated with the label *lbl*. This does not destroy the stream generator. If no generator is associated with the label then nothing is done.

- `get_stream_generator`(*lbl*): Return the stream generator associated with the label *lbl* if it exists, otherwise throw a NoSuchLabel exception.

5.4.3 Streams

When implementing streams an important requirement is that the consumption of samples should be decoupled from the production of samples in the sense that the producer should be able to continue producing new samples disregarding how long a consumer takes to process a sample. It should also be possible to destroy a stream even if there are clients connected and to replace a stream where the connected clients will get the content of the new stream without having to reconnect. The motivation behind these requirements is that we are working in a distributed system where the different parts should be as decoupled as possible. Clients and hosts might for example become unavailable either permanently or temporarily which should not affect the DyKnow service itself or those parts of the application that do not use these clients or hosts.

To decouple stream generators from clients the DyKnow service uses an event channel to distribute the content of streams. There are at least two approaches to implementing streams using an event channel. The first is to follow the conceptual model closely by letting the stream generator create one new stream for each policy on the host side, adapting it to the associated policy, and pushing every element in each stream onto the event channel. This would make it easy for the client since it only has to give the policy to the stream generator and then listen for the samples

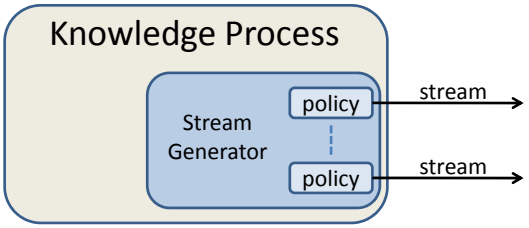


Figure 5.4: A conceptual view of how a single stream generator supports many different subscriptions to the output created by its knowledge process.

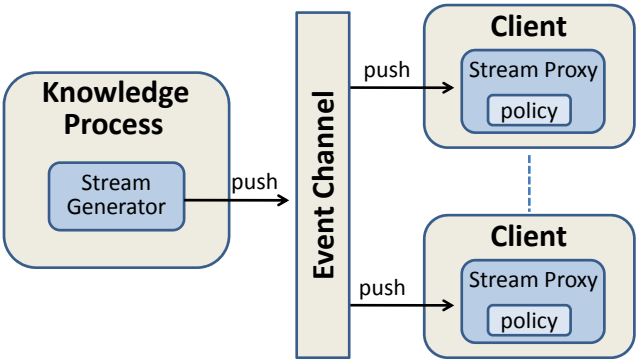


Figure 5.5: An overview of how several streams are created from a single stream generator and distributed to the clients using the event channel.

as they arrive. The downside is that the policy would only be satisfied on the stream generator side of the event channel. If the event channel reorders samples, introduces delays, or even loses samples then the stream as seen by the client will be different compared to the stream as generated by the stream generator.

The solution to this issue is to implement the policy on the client side. A stream generator gets a stream of output from a knowledge process. Each element in this stream is then pushed, as soon as it is generated, on the event channel tagged with the label of the stream generator. A client interested in the output of the knowledge process creates a stream proxy from a label and a policy. The stream proxy subscribes to all samples with the label pushed on the event channel. The sequence of samples received by the client is then adapted according to the policy to generate a local stream which satisfies the policy.

This means that the conceptual view of a stream generator taking policies and generating a stream for each of them (Figure 5.4) is replaced with a stream generator pushing stream elements on an event channel and clients subscribing to these elements and adapting them locally according to their policies (Figure 5.5).

The event channel implementation of streams decouples stream generators from clients and supports asynchronous push-based delivery of samples.

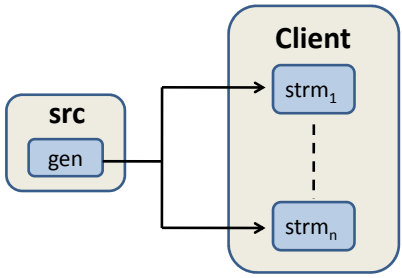


Figure 5.6: The experimental setup when varying the number of fluent streams.

5.5 Empirical Evaluation

To evaluate the performance and the scalability of the current DyKnow implementation a number of experiments are performed. In a knowledge processing application most of the time will be spent in the knowledge processes, such as image processing, fusion of data from IMU and GPS, and chronicle recognition. These processes are application specific and may vary considerably. Therefore we choose to measure the time spent in DyKnow generating stream content by stream generators, sending samples over the CORBA event channel to multiple clients, and processing them on the client side by stream proxies.

To evaluate the performance of DyKnow, we therefore measure how delays are influenced when varying the number of concurrent streams and knowledge processes in a knowledge processing application. There are two types of delays, total delays and notification channel delays. The total delay is the difference between the time when a sample is available at a client and the valid time of the sample. The notification channel delay is the time spent in the event channel. Three experiments are performed. In the experiments, we vary the number of concurrent fluent streams, sources, and computational units respectively. These experiments provide an insight into the scalability of the current implementation. It should be noted that the implementation is not optimized and it is designed for a distributed system where sources and computational units can be hosted on different machines.

In all experiments only a single computer is used, one of the computers onboard our UAV, a PC104 Pentium-M 1.4 GHz embedded computer with 1 GB RAM. The main reason we keep everything on the same machine is to measure the overhead introduced by CORBA and DyKnow as opposed to the network. However, since we use CORBA it is easy to distribute a knowledge processing application over many machines in order to handle large, complex, and computationally demanding applications.

Varying the Number of Fluent Streams

The first experiment studies the effect on the delay when varying the number of fluent streams generated by a source. In each iteration of this experiment there

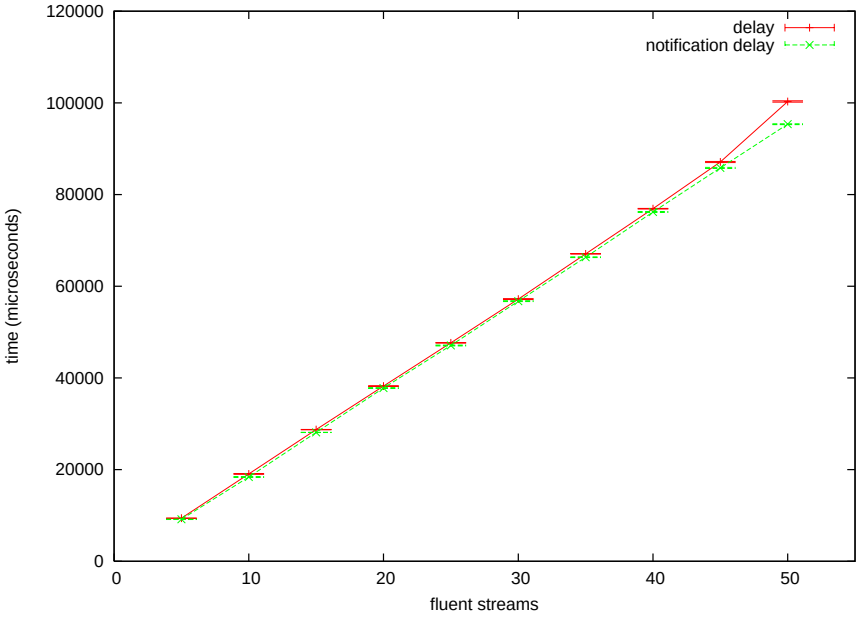


Figure 5.7: The total delay and the notification channel delay when varying the number of fluent streams generated from a single source.

is a source and a client subscribing to n fluent streams generated from the source according to the same policy (Figure 5.6). The output provided by the fluent stream generator of the source contains a new sample every 100 milliseconds. The KPL specification of the experiment application for a given number of streams n is:

source `int` src
strmgen gen = src
stream strm_1 = gen **with sample every** 100
...
stream strm_n = gen **with sample every** 100

In the case where n streams are used, each sample produced by the source is sent once to the event channel, after which the client needs to receive this sample through n separate streams. Samples are sent every 100 milliseconds during one minute, for a total of 600 samples. Since only one CPU is available, reception will necessarily take place in some sequential order. The stream that gets a sample first will have a very short delay, while the stream that gets the sample last will have a longer delay. This means that the average delay over the 600 samples for each stream will vary depending on its place in the sequential order. Since we are interested in ensuring that all clients receive samples with sufficient speed even in the worst case we defined the delay for an iteration as the highest average delay among the n streams. Because the current implementation delivers samples in a deterministic order, this is equivalent to the average delay for the stream that

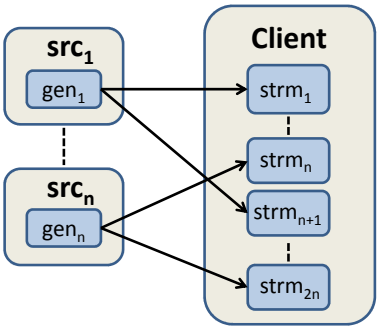


Figure 5.8: The experimental setup when varying the number of sources in the case where the number of sources is n and the number of streams is $2n$.

always receives its samples last.

We repeat this entire experiment 10 times. The final result reported for n streams is the average of the results from the 10 iterations. The same is done for the notification channel delay.

The result of varying the number of fluent streams from 5 to 50 with an increment of 5 streams is shown in Figure 5.7. The graph shows that the total delay and the notification channel delay linearly increase with the number of streams. The delay is approximately 2 milliseconds per stream. The graph also shows that the delay is almost entirely due to the notification channel overhead, which does not depend on DyKnow but on the particular CORBA event channel implementation being used. In other words, DyKnow itself appears quite efficient, and any improvements in overall performance would most likely have to be achieved by replacing the underlying event channel mechanism.

Varying the Number of Sources

In the second experiment we study the effect on the delay when fixing the number of streams and varying the number of sources. In each iteration of this experiment there are n sources and a client subscribing to m fluent streams generated by the sources (Figure 5.8). If $m > n$ then the same source provides input to more than one stream. The number of fluent streams m is fixed while the number of sources n varies. The output provided by the fluent stream generator of each source contains a new sample every 100 milliseconds. The KPL specification of the experiment application for n sources and m streams is:

source `int` src_1
strmgen $\text{gen}_1 = \text{src}_1$
...
source `int` src_n
strmgen $\text{gen}_n = \text{src}_n$
stream $\text{strm}_1 = \text{gen}_1$ **with sample every** 100

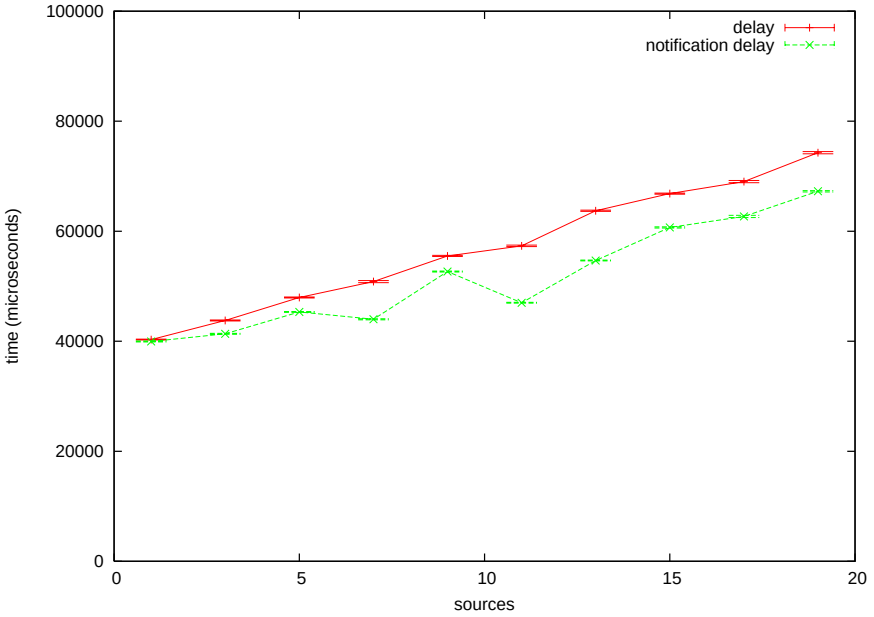


Figure 5.9: The total delay and the notification channel delay when fixing the number of streams and varying the number of sources.

...
stream strm$_n$ = gen$_n$ **with sample every** 100
stream strm$_{n+1}$ = gen$_1$ **with sample every** 100
...
stream strm$_m$ = gen$_{((m-1)\,mod\,n)+1}$ **with sample every** 100

In the case where n sources and m streams are used, each sample produced by a source is sent once to the event channel, after which the client needs to receive this sample through at most $\lceil m/n \rceil$ separate streams. Samples are sent every 100 milliseconds during one minute, for a total of 600 samples. All sources produce samples at the same valid times in order to measure the worst case delay instead of the average delay. Like in the previous experiment, we only use a single CPU and the reception will therefore necessarily take place in some sequential order. To measure the worst case delay of any of the m streams, the delay for an iteration is defined as the highest average delay among the streams. Because the current implementation delivers samples in a deterministic order, this is equivalent to the average delay for the stream that always receives its samples last.

We repeat this entire experiment 10 times. The final result reported for n sources is the average of the results from the 10 iterations. The same is done for the notification channel delay.

The result when fixing the number of streams to 19 and varying the number of sources from 1 to 19 with an increment of 2 sources is shown in Figure 5.9. As can

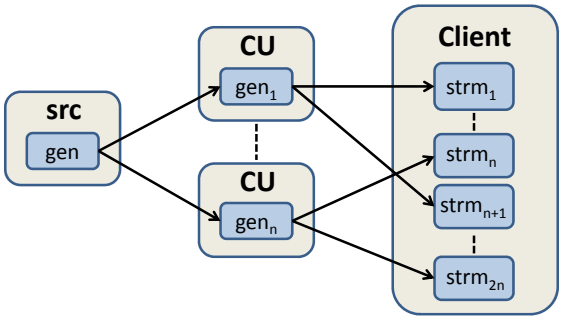


Figure 5.10: The experimental setup when varying the number of computational units in the case when the number of computational units is n and the number of streams is $3n$ (n streams are used to provide input to the computational units).

be seen from the graph the maximum delay increase linearly with the number of sources. The approximated delay per source is 1 millisecond, and the initial delay introduced by the 19 streams is about 40 milliseconds. As with the previous experiment, the total delay is almost entirely due to the notification channel overhead, which does not depend on DyKnow but on the particular CORBA event channel implementation being used.

Varying the Number of Computational Units

In the third experiment we study the effect on the delay when fixing the number of streams and varying the number of computational units. In each iteration of this experiment there is a source, n computational units, and a client subscribing to m fluent streams generated by the computational units (Figure 5.10). The computational unit takes a single stream as input and provides a copy of each input sample as output. Of the m fluent streams, n will be used as input to the computational units. A client will subscribed to the remaining $m - n$ streams. If $m - n > n$ then the same computational unit provides input to more than one stream subscribed to by the client. The number of fluent streams m is fixed while the number of computational units n varies. The input fluent streams are all generated from the same source with the policy "**sample every** 100". The output provided by the fluent stream generator of the source contains a new sample every 100 milliseconds. The KPL specification of the experiment application for a given number of computational units n and streams m is:

source `int` src
compunit `int` CU(`int`)
strmgen gen = src
strmgen gen_1 = CU(gen **with sample every** 100)
...
strmgen gen_n = CU(gen **with sample every** 100)

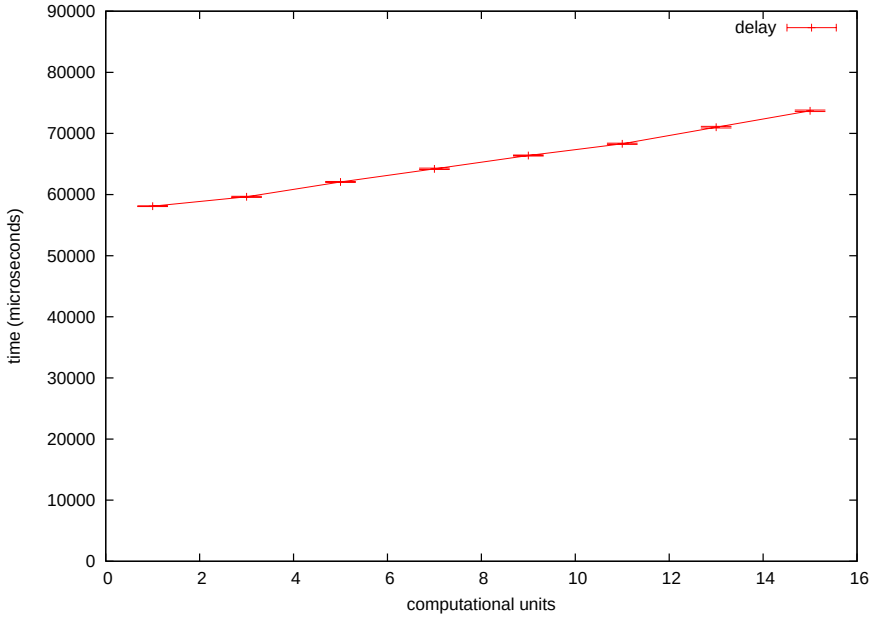


Figure 5.11: The total delay when fixing the number of streams and varying the number of computational units using a single source.

stream $\mathsf{strm}_1 = \mathsf{gen}_1$ **with sample every** 100
...
stream $\mathsf{strm}_n = \mathsf{gen}_n$ **with sample every** 100
stream $\mathsf{strm}_{n+1} = \mathsf{gen}_1$ **with sample every** 100
...
stream $\mathsf{strm}_{m-n} = \mathsf{gen}_{((m-n-1) \bmod n)+1}$ **with sample every** 100

In the case where n computational units and m streams are used, each sample produced by the single source is sent once to the event channel, after which the n computational units need to receive this sample. When a computational unit receives a sample it will make a copy of it available to its stream generator. The stream generator will push the new sample to the event channel where the client needs to receive it through at most $\lceil (m-n)/n \rceil$ separate streams.

Samples are sent by the source every 100 milliseconds during one minute, for a total of 600 samples. All samples produced by the computational units will retain the valid time from the sample received from the source. It is therefore possible to measure the worst case delay for every sample originating from the source. Like in the previous experiment, we only use a single CPU and the reception will therefore necessarily take place in some sequential order. To measure the worst case delay of any of the m streams, the delay for an iteration is defined as the highest average delay among the streams. Because the current implementation delivers samples in a deterministic order, this is equivalent to the average delay for the stream that

always receives its samples last.

We repeat this entire experiment 10 times. The final result reported for n computational units is the average of the results from the 10 iterations. We do not measure the notification channel delay for this experiment since it is hard to single out from the total delay.

The result when fixing the number of streams to 30 and varying the number of computational units from 1 to 15 with an increment of 2 computational units is shown in Figure 5.11. As can be seen from the graph the maximum delay increases linearly with the number of computational units.

5.6 Summary

This chapter has described a DyKnow middleware service supporting the implementation of knowledge processing applications specified in KPL. The service is built as a CORBA service which leverages its support for distributed systems. This means that applications developed using the DyKnow service can be distributed over many different computers and programming languages. This is very useful in network centric systems or advanced autonomous systems where more than one computer is used.

The core DyKnow service has three components. The first is the event channel that provides asynchronous many-to-many communication between clients and knowledge processes in order to decouple them. The second is the Knowledge Process Factory that keeps track of the sources and parameterized computational units available in the application and creates instances of these. This means that a client does not need to know which host actually creates the knowledge process whose output it subscribes to. The final component is the Stream Generator Manager which keeps track of all the stream generators in the application. Each stream generator is associated with a label which can be used by a client either to request a query interface to the stream, or to connect to the event channel to incrementally get samples as they are produced. This means that if a stream generator has already been created it is enough for a client to know its label to be able to access it. This is another benefit of decoupling since the creation of a knowledge process can be done by one node in the distributed system while other nodes only need to know the label of the stream generator to get its output.

The DyKnow middleware service provides interfaces that can be used by a host to integrate any knowledge process into a DyKnow knowledge processing application. The DyKnow service is not dependent on the actual implementation, as long as the stream generator interface is implemented. This allows application specific implementations of sources, computational units, and stream generators.

The DyKnow middleware service provides a flexible and capable implementation of the DyKnow middleware framework suitable for distributed real-time and embedded systems such as autonomous UAV systems.

Part III

Applications and Extensions

Chapter 6

The UASTech UAV Platform

6.1 Introduction

As stated in the introduction, one important application area for knowledge processing is the emerging area of intelligent unmanned aerial vehicle (UAV) research, which has shown rapid development in recent years and offers a great number of research challenges. Much previous research has focused on low-level control capabilities with the goal of developing controllers which support the autonomous flight of a UAV from one way-point to another. A common type of mission scenario involves placing sensor payloads in position for data collection tasks where the data is eventually processed off-line or in real-time by ground personnel. The use of UAVs and mission tasks such as these have become increasingly more important in recent conflict situations and are predicted to play increasingly more important roles in any future conflicts. Intelligent UAVs will play an equally important role in civil applications.

For both military and civil applications, there is a desire to develop more sophisticated UAV platforms where the emphasis is placed on the development of intelligent capabilities and on abilities to interact with human operators and additional robotic platforms. The focus in this research has moved from low-level control towards a combination of low-level and decision-level control integrated in sophisticated software architectures. These, in turn, should also integrate well with larger network-centric based C^4I^2 (Command, Control, Communications, Computers, Intelligence, Interoperability) systems. Such platforms are a prerequisite for supporting the capabilities required for the increasingly more complex mission tasks on the horizon and provide an ideal testbed for the development and integration of distributed AI technologies.

For a number of years, The Autonomous Unmanned Aircraft Systems Technologies Lab[1] (UASTech Lab) at Linköping University, Sweden, has pursued a long term research endeavor related to the development of future aviation systems in the form of autonomous unmanned aerial vehicles (Doherty et al., 2000; Do-

[1] The UASTech Lab was previously called the UAVTech Lab.

Figure 6.1: The UASTech Yamaha RMAX helicopter.

herty, 2004, 2005). The focus has been on both high autonomy (AI related functionalities), low level autonomy (traditional control and avionics systems), and their integration in distributed software architectural frameworks (Doherty et al., 2004) in order to support robust autonomous operation in complex operational environments such as those one would face in catastrophe situations. Some existing application scenarios are traffic monitoring and surveillance, emergency services assistance, and photogrammetry and surveying where the first two were described in Sections 1.1.1 and 1.1.2.

Basic and applied research in the project covers a wide range of topics which include the development of a distributed architecture for autonomous unmanned aerial vehicles. In developing the architecture, the larger goals of integration with human operators and other ground and aerial robotics systems in network-centric C^4I^2 infrastructures have been taken into account and influenced the nature of the base architecture. In addition to the software architecture and the knowledge processing middleware component, several AI technologies have been developed such as path planners (Pettersson, 2006; Wzorek and Doherty, 2009; Wzorek et al., 2006), a task planner (Kvarnström, 2005), and the execution monitoring and chronicle recognition functionalities described in Chapters 7 and 8.

More recently, our research has moved from single platform scenarios to multi-platform scenarios where a combination of UAV platforms with different capabilities are used together with human operators in a mixed-initiative context with adjustable platform autonomy (Doherty and Meyer, 2007).

6.2 UAV Platforms and Hardware Architecture

The UASTech UAV platform (Doherty, 2004) is a slightly modified Yamaha RMAX helicopter (Figure 6.1). It has a total length of 3.6 m (including the main rotor) and

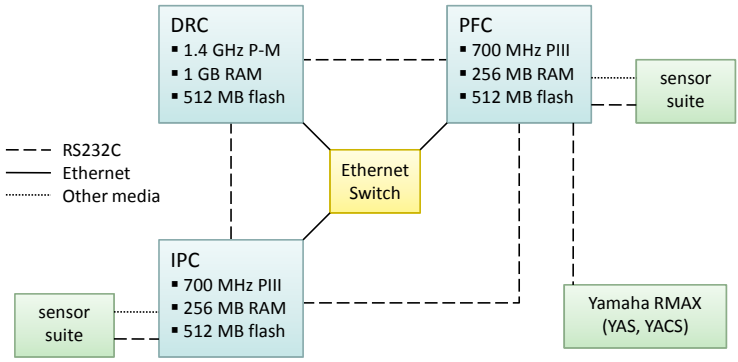


Figure 6.2: Onboard hardware schematic.

is powered by a 21 hp two-stroke engine with a maximum takeoff weight of 95 kg. Our hardware platform is integrated with the Yamaha platform as shown in Figure 6.2. It contains three PC104 embedded computers.

The primary flight control (PFC) system runs on a 700 MHz PIII, and includes a wireless Ethernet bridge, an RTK GPS receiver, and several additional sensors including a barometric altitude sensor. The PFC is connected to the Yamaha Attitude Sensors (YAS), the Yamaha Attitude Control System (YACS), an image processing computer, and a computer for deliberative capabilities.

The image processing system (IPC) runs on a second PC104 embedded Pentium III 700 MHz computer. The camera platform suspended under the UAV fuselage is vibration isolated by a system of springs. The platform consists of a Sony FCB-780P CCD block camera and a ThermalEye-3600AS miniature infrared camera mounted rigidly on a pan-tilt unit as shown in Figure 6.3. The video footage from both cameras is recorded at full frame rate by two MiniDV recorders to allow postprocessing after flights.

The deliberative/reactive system (DRC) runs on a third PC104 Pentium-M 1.4 GHz embedded computer and executes all high-end autonomous functionality. Network communication between computers is physically realized with serial line RS232C and Ethernet. Ethernet is mainly used for CORBA applications, remote login, and file transfer, while serial lines are used for hard real-time networking.

More recently, we have developed a number of micro aerial vehicles (Duranti et al., 2007; Rudol et al., 2008) for our experimentation with cooperative UAV systems. The intent is to use these together with our RMAX systems for cooperative missions.

6.3 The Software System Architecture

A hybrid deliberative/reactive software architecture has been developed for our RMAX UAVs. Conceptually, it is a layered, hierarchical system with deliberative,

Figure 6.3: The UASTech UAV and the onboard camera system mounted on a pan-tilt unit.

reactive, and control components. Figure 6.4 presents the functional layer structure of the architecture and emphasizes its reactive-concentric nature.

With respect to timing characteristics, the architecture can be divided into two layers: (a) the hard real-time part, which mostly deals with hardware and control laws (also referred to as the Control Kernel) and (b) the non real-time part, which includes deliberative services of the system (also referred to as the High-Level System)[2].

All three computers in our UAV platform (PFC, IPC, and DRC) have both hard and soft real-time components but the processor time is assigned to them in different proportions. On one extreme, the PFC runs mostly hard real-time tasks with only minimum user space applications (e.g. SSH daemon for remote login). On the other extreme, the DRC uses the real-time part only for device drivers and real-time communication. The majority of processor time is spent on running the deliberative services.

The Control Kernel (CK) is a distributed real-time runtime environment and is used for accessing the hardware, implementing continuous control laws, and controlling mode switching. Moreover, the CK coordinates the real-time communication between all three onboard computers as well as between CKs of other robotic systems. In our case, we perform multi-platform missions with two identical RMAX helicopter platforms. The CK is implemented using C code. This part of the system uses the Real-Time Application Interface (RTAI) (Mantegazza *et. al.*, 2000) which provides industrial-grade real-time operating system functionality. RTAI is a hard real-time extension to a standard Linux kernel (Debian in our case) and has been developed at the Department of Aerospace Engineering of Politecnico di Milano.

The real-time performance is achieved by inserting a module into the Linux kernel space. Since the module takes full control over the processor it is necessary to suspend it in order to let the user space applications run. The standard Linux

[2]Note that the distinction between the Control Kernel and the High-Level System is conceptual and based mainly on timing characteristics; it does not exclude, for example, placing deliberative services (e.g. prediction) in the Control Kernel.

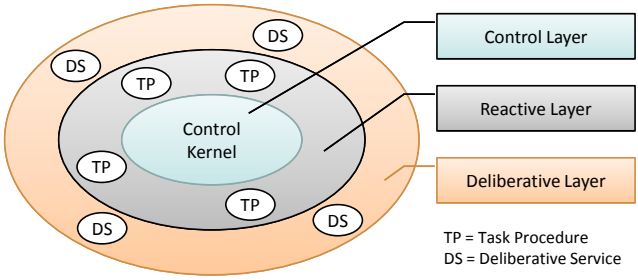


Figure 6.4: The functional structure of the architecture.

distribution is a task with lower priority, which is run preemptively and can be interrupted at any time. For that reason a locking mechanism is used when both user- and kernel-space processes communicate through shared memory. It is also important to mention that the CK is self-contained and only the part running on the PFC computer is necessary for maintaining flight capabilities. Such separation enhances safety of the operation of the UAV platform which is especially important in urban environments.

The Control Kernel has a hybrid flavor. Its components contain continuous control laws and mode switching is realized using event-driven hierarchical concurrent state machines (HCSMs) (Merz, Rudol, and Wzorek, 2006). HCSMs can be represented as state transition diagrams and are similar to statecharts (Harel, 1987). In our system, tables describing transitions derived from such diagrams are passed to the system in the form of text files and are interpreted by a HCSM interpreter at run-time on each of the onboard computers. Thanks to its compact and efficient implementation, the interpreter runs in the real-time part of the system as a task with high execution rate. It allows coordinating all *functional units* of the control system from the lowest level hardware components (e.g. device drivers) through control laws (e.g. hovering and path following) and communication to the interface used by the Helicopter Server.

The use of HCSMs also allows implementing complex behaviors consisting of other lower level ones. For instance, the landing mode includes control laws steering the helicopter and coordinating camera system/image processing functionalities. When the landing behavior is activated, the CK takes care of searching for a pre-defined pattern with the camera system, feeding a Kalman filter with image processing results which fuses them with the helicopter's inertial measurements. The CK sends appropriate feedback when the landing procedure is finished or it has been aborted. For details see Merz, Duranti, and Conte (2004).

For achieving the best performance, a single non-preemptive real-time task is used which follows a predefined static schedule to run all functional units. Similarly, the real-time communication physically realized using serial lines is statically scheduled with respect to packet sizes and rates of sending. For a detailed description see Merz (2004).

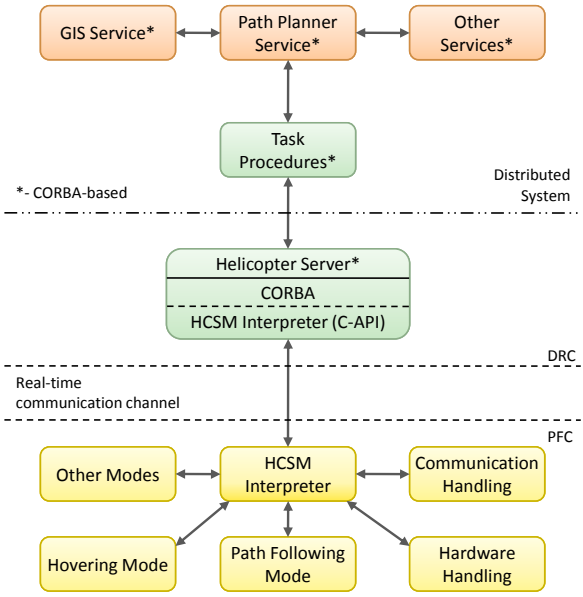


Figure 6.5: The main software components of the navigation subsystem.

The high-level part of the system has reduced timing requirements and is responsible for coordinating the execution of reactive Task Procedures (TPs). A TP is a high-level procedural execution component which provides a computational mechanism for achieving different robotic behaviors by using both deliberative services and traditional control components in a highly distributed and concurrent manner. The control and sensing components of the system are accessible for TPs through the Helicopter Server which in turn uses an interface provided by the Control Kernel. A TP can initiate one of the autonomous control flight modes available in the UAV (e.g. take off, vision-based landing, hovering, dynamic path following, or reactive flight modes for interception and tracking). An overview of the navigation subsystem is shown in Figure 6.5. Additionally, TPs can control the payload of the UAV platform which currently consists of video and thermal cameras mounted on a pan-tilt unit in addition to a stereo camera system. TPs can also receive helicopter state delivered by the PFC computer and camera system state delivered by the IPC computer, including image processing results.

The software implementation of the high-level system is based on CORBA (Common Object Request Broker Architecture), which is often used as middleware for object-based distributed systems. It enables different objects or components to communicate with each other regardless of the programming languages in which they are written, their location on different processors, or the operating systems they are running in. A component can act as a client, a server, or as both. The functional interfaces to components are specified via the use of IDL (Interface

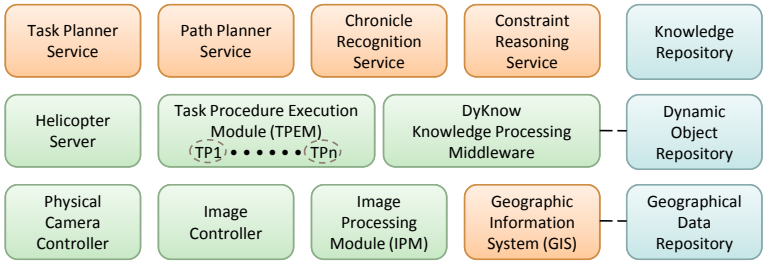


Figure 6.6: Some deliberative, reactive and control services.

Definition Language). The majority of the functionalities which are part of the architecture can be viewed as CORBA objects or collections of objects, where the communication infrastructure is provided by CORBA facilities and other services such as real-time and standard event channels.

This architectural choice provides us with an ideal development environment and versatile run-time system with built-in scalability, modularity, software relocatability on various hardware configurations, performance (real-time event channels and schedulers), and support for plug-and-play software modules.

Figure 6.6 presents some (not all) of the high-level services used in the UASTech UAV system including the DyKnow knowledge processing middleware. Those services run on the deliberative/reactive computer and interact with the control system through the Helicopter Server. The Helicopter Server on one side uses CORBA to be accessible by TPs or other components of the system; on the other side it communicates through shared memory with the HCSM based interface running in the real-time part of the DRC software.

6.4 Conclusions

The UAV platform presented in this chapter provides an appropriate vehicle where the full potential of DyKnow can be used to develop challenging scenarios which can be tested in a dynamic environment. In order to attack more complex problems the platform must provide powerful and sophisticated functionality and have reached a certain level of maturity. The platform presented has been extensively tested both in simulation and in live test flights in a small urban area.

Chapter 7

Integrating Planning and Execution Monitoring

This chapter contains an article *A Temporal Logic-based Planning and Execution Monitoring Framework for Unmanned Aircraft Systems* which has been extended with a significantly expanded section on state generation (Section 7.8). The article is accepted for publication in the Journal of Automated Agents and Multi-Agent Systems (Doherty, Kvarnström, and Heintz, 2009) published by Springer. To make the article fit the rest of the thesis, Sections 2–4 in the original article have been removed since the material is already covered mainly in Chapter 6, Section 5 is incorporated in the modified introduction, and Section 4.2 is lifted up to its own section. Apart from the restructuring and the greatly extended state generation section no significant changes have been made.

7.1 Introduction

Now and then, things will go wrong. This is both a fact of life and a fundamental problem in any robotic system intended to operate autonomously or semi-autonomously in the real world. Like humans, robotic systems (or more likely their designers) must be able to take this into account and recover from failures, regardless of whether those failures result from mechanical problems, incorrect assumptions about the world, or interference from other agents.

In this chapter, we present a temporal logic-based task planning and execution monitoring framework and its integration into the fully deployed rotor-based unmanned aircraft system described in the previous chapter. In the spirit of cognitive robotics, we make specific use of Temporal Action Logic (TAL (Doherty and Kvarnström, 2008)), a logic for reasoning about action and change. This logic has already been used as the semantic basis for a task planner called TALplanner (Kvarnström, 2005), which is used to generate mission plans that are carried out by an execution subsystem.

We show how knowledge gathered from the appropriate sensors during plan execution can be used by DyKnow to incrementally create state structures. These state structures correspond to partial logical models in TAL, representing the actual development of the system and its environment over time. We then show how formulas in TAL can be used to specify the desired behavior of the system and its environment and how violations of such formulas can be detected in a timely manner in an execution monitor subsystem which makes use of a progression algorithm for prompt failure detection.

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

Experimentation with the system has been done in the context of the challenging emergency services scenario introduced in Section 1.1.2, involving body identification of injured civilians on the ground and logistics missions to deliver medical and food supplies to the injured (Doherty and Rudol, 2007) using several UAVs in a cooperative framework (Doherty and Meyer, 2007). A combination of real missions flown with our platforms and hardware-in-the-loop simulations has been used to verify the practical feasibility of the proposed systems and techniques. We will now describe the two parts of the scenario in more detail.

7.1.1 Mission Leg I : Body Identification

The task of the 1st leg of the mission is to scan a large region with one or more UAVs, identify injured civilians, and output a saliency map which can be used by emergency services or other UAVs. Our approach (Doherty and Rudol, 2007; Rudol and Doherty, 2008) uses two video sources (thermal and color) and allows for high rate human detection at larger distances than in the case of using the video sources separately with standard techniques. The high processing rate is essential in case of video collected onboard a UAV in order not to miss potential victims as a UAV flies over them.

A thermal image is analyzed first to find human body sized silhouettes. Corresponding regions in a color image are subjected to a human body classifier which is configured to allow weak classifications. This focus of attention allows for maintaining a body classification at a rate up to 25 Hz. This high processing rate allows for collecting statistics about classified humans and to prune false classifications of the ”weak” human body classifier. Detected human bodies are geolocalized on a map which can be used to plan supply delivery. The technique presented has been tested onboard the UASTech UAV platform and is an important component in our research with autonomous search and rescue missions.

Experimental setup

A series of flight tests were performed in southern Sweden at an emergency services training center used by the Swedish Rescue Services Agency to train fire, po-

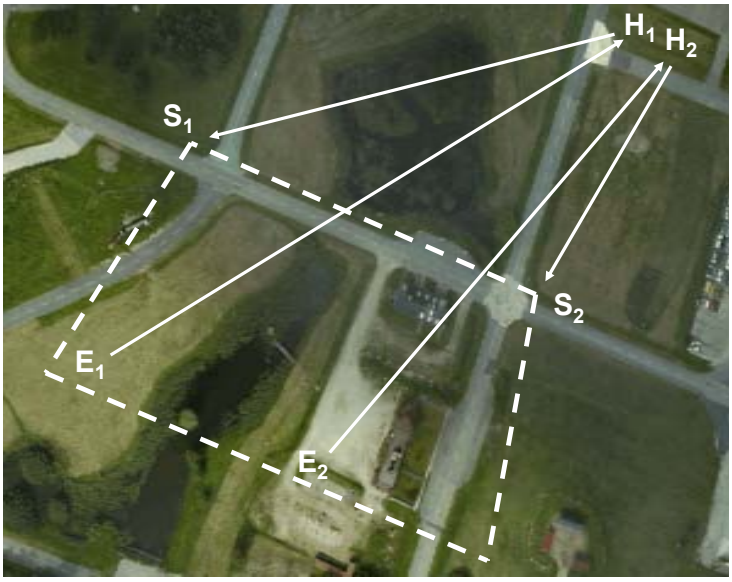


Figure 7.1: Mission overview.

lice, and medical personnel. This area consists of a collection of buildings, roads, and even makeshift car and train accidents. This provides an ideal test area for experimenting with traffic surveillance, photogrammetric, and surveying scenarios, in addition to scenarios involving emergency services. We have also constructed an accurate 3D model for this area which has proven invaluable in simulation experiments and parts of which have been used in the onboard GIS (Geographic Information System).

Flight tests were performed over varied terrain such as asphalt and gravel roads, grass, trees, water, and building roof tops which resulted in a variety of textures in the images. Two UAVs were used over a search area of 290×185 meters. A total of eleven bodies (both human and dummies with close to human temperature) were placed in the area. The goal of the mission was to generate a saliency map. The general mission plan is shown in Figure 7.1.

Before take-off, one of the UAVs was given an area to scan (dashed line polygon). It then delegated part of the scanning task to another platform, generating sub-plans for itself and the other platform. The mission started with a simultaneous autonomous take-off at positions H_1 and H_2 and the UAVs flew to starting positions S_1 and S_2 for scanning. Throughout the flights, saliency maps were incrementally constructed until the UAVs reached their ending positions E_1 and E_2. The UAVs then returned to their respective take-off positions for a simultaneous landing. The mission took approximately ten minutes to complete and each UAV traveled a distance of around 1 km.

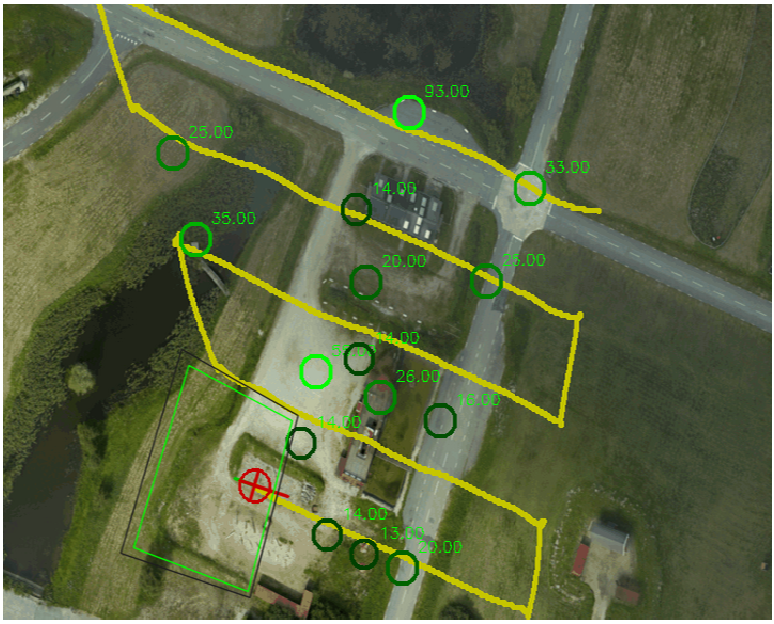


Figure 7.2: Flight path and geolocated body positions.

Experimental results

The algorithm found all eleven bodies placed in the area. The saliency map generated by one of the helicopters is shown in Figure 7.2. The images of identified objects are presented in Figure 7.3 on the next page. Several positions were rejected as they were not observed long enough (i.e. 5 seconds). Images 7, 9, and 14 present three falsely identified objects.

7.1.2 Mission Leg II: Package Delivery

After successful completion of leg I of the mission scenario, we can assume that a saliency map has been generated with geo-located positions of the injured civilians. In the next phase of the mission, the goal is to deliver configurations of medical, food, and water supplies to the injured. In order to achieve this leg of the mission, one would require a task planner to plan for logistics, a motion planner to get one or more UAVs to supply and delivery points, and an execution monitor to monitor the execution of highly complex plan operators. Each of these functionalities would also have to be tightly integrated in the system.

This leg of the mission will be used as a running example for the rest of this chapter. For these logistics missions, we assume the use of one or more UAVs with diverse roles and capabilities. Initially, we assume there are n injured body locations, several supply depots, and several supply carrier depots (see Figure 7.5 on page 93). The logistics mission is comprised of one or more UAVs transport-

Figure 7.3: Images of classified bodies – corresponding thermal images are placed under color images.

ing boxes containing food and medical supplies between different locations (Figure 7.4).

Achieving the goals of such a logistics mission with full autonomy requires the ability to pick up and deliver boxes without human assistance; thus, each UAV has a device for attaching to boxes and carrying them beneath the UAV. The action of picking up a box involves hovering above the box, lowering the device, attaching to the box, and raising the device, after which the box can be transported to its destination. There can also be a number of carriers, each of which is able to carry several boxes. By loading boxes onto such a carrier and then attaching to the carrier, the transportation capacity of a UAV increases manyfold over longer distances. The ultimate mission for the UAVs is to transport the food and medical supplies to their destinations as efficiently as possible using the carriers and boxes at their disposal.

An attachment device consisting of a winch and an electromagnet is under development. In the mean time, the logistics scenario has been implemented and tested in a simulated UAV environment with hardware in-the-loop, where TALplanner generates a detailed mission plan which is then sent to a simulated execution system using the same helicopter flight control software as the physical UAV. Each UAV has an execution monitor subsystem which continuously monitors its operation in order to detect and signal any deviations from the declaratively specified intended behavior of the system, allowing the main execution system to initiate the appropriate recovery procedures. The information needed by the execution monitoring system is collected and synchronized by DyKnow. Faults can be injected through the simulation system, enabling a large variety of deviations to be tested. Additionally, the simulator makes use of the Open Dynamics Engine[1], a library for simulating rigid body dynamics, in order to realistically emulate the physics of boxes and carriers. This leads to effects such as boxes bouncing and rolling

[1] http://www.ode.org

Figure 7.4: The UAV logistics simulator.

Figure 7.5: A supply depot (left) and a carrier depot (right).

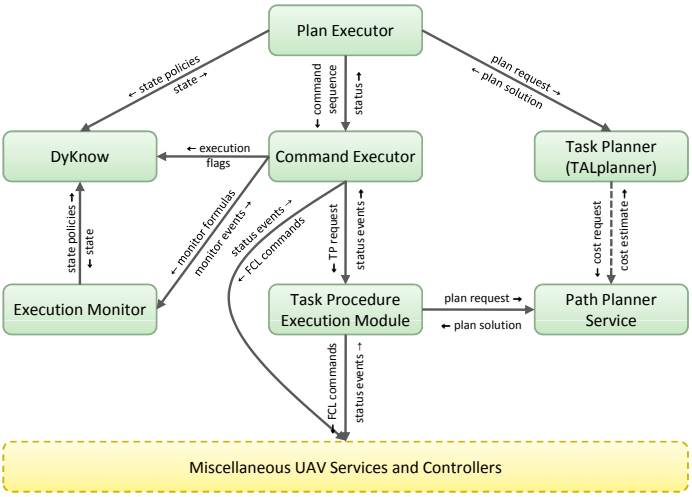


Figure 7.6: Task planning and execution monitoring overview.

away from the impact point should they accidentally be dropped, which is also an excellent source of unexpected situations that can be used for validating both the domain model and the execution monitoring system.

The use of TALplanner to generate formulas, the collection and synchronization of information using DyKnow, the on-line evaluation of these formulas by the execution monitoring subsystem, and the recovery of detected formula violations have been tested in real flight tests using a photogrammetry mission which does not require a winch system.

7.2 Task Planning and Execution Monitoring System Overview

The main part of the architecture we will focus on for the remainder of the chapter involves those components associated with task planning, execution of task plans, and execution monitoring. Figure 7.6 shows the relevant part of the UAV system architecture associated with these components.

At the top of the center column is the *plan executor* which given a mission request calls DyKnow to acquire essential information about the current contextual state of the world or the UAV's own internal states. Together with a domain specification and a goal specification related to the current mission, this information is fed to *TALplanner* (Doherty and Kvarnström, 2001, 2008; Kvarnström, 2005), a logic-based task planner which outputs a plan that will achieve the designated goals, under the assumption that all actions succeed and no failures occur. Such a plan can also be automatically annotated with global and/or operator-specific conditions to be monitored during execution of the plan by an execution monitor in order to

relax the assumption that no failures can occur. Such conditions are expressed as temporal logical formulas and evaluated on-line using formula progression techniques. The execution monitor notifies the command executor when actions do not achieve their desired results and one can then move into a plan repair phase.

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

Task Procedures can use a path planner to generate collision free trajectories through the environment. Currently, we use a number of different techniques for path planning (Wzorek and Doherty, 2009; Wzorek et al., 2006) which include the use of Probabilistic Roadmaps (Kavraki et al., 1996) and Rapidly Exploring Random Trees (Kuffner and LaValle, 2000). For the purposes of this chapter, we will simply assume that our UAVs can automatically generate and fly collision free trajectories from specified start and end points in operational environments.

During plan execution, the command executor adds formulas to be monitored to the *execution monitor*. DyKnow continuously sends information about the development of the world in terms of state sequences to the monitor, which uses a progression algorithm to partially evaluate monitor formulas. If a violation is detected, this is immediately signaled as an event to the command executor, which can suspend the execution of the current plan, invoke an emergency brake command if required, optionally execute an initial recovery action, and finally signal the new status to the plan executor. The plan executor is then responsible for completing the recovery procedure (Section 7.5.3 on page 112).

The fully integrated system is implemented on our UAVs and can be used onboard for different configurations of the logistics mission described in Leg II of the larger mission. The simulated environments used are in urban areas and quite complex. Plans are generated in the millisecond to seconds range using TALplanner and empirical testing shows that this approach is promising in terms of integrating high-level deliberative capability with lower-level reactive and control functionality.

7.3 Background: Temporal Action Logic

This section introduces TAL (Temporal Action Logic), a framework used for reasoning about action and change. More specifically, this section will focus on the use of TAL-C (Doherty and Kvarnström, 2008; Karlsson and Gustafsson, 1999), one of the more recent members of the TAL family of logics with support for concurrent actions. Though no new results will be presented here, a basic understanding of TAL will be useful when reading other parts of this chapter. We refer the reader to Doherty and Kvarnström (2008) or Doherty et al. (1998) for further details.

An agent using TAL is assumed to be interested in one or more reasoning tasks, such as prediction or planning, related to a specific *world* such as the UAV domain. It is assumed that the world is dynamic, in the sense that the various properties or *features* of the world can change over time. TAL-C is an order-sorted logic where each feature, as well as each of its parameters, is specified to take values in a specific *value domain*. The value domain `boolean` = {**true**, **false**} is always assumed to be present. In addition, the UAV domain could define the two value domains `uav` = {**heli1**, **heli2**} for UAVs and `box` = {**bx1**, **bx2**, **bx3**, **bx4**} for boxes to be delivered. The parameterized boolean-valued feature attached(*uav*, *box*) could then represent the fact that *uav* has picked up *box*, while the integer-valued feature capacity(*uav*) could be used to model the carrying capacity of *uav*.

TAL offers a modular means of choosing temporal structures depending on the nature of the world being reasoned about and the reasoning abilities of the agent. TAL-C is based on the use of a linear (as opposed to branching) discrete non-negative integer time structure where time 0 corresponds to the initial state in a reasoning problem. The temporal sort is assumed to be interpreted, but can be axiomatized in first-order logic as a subset of Presburger arithmetic, natural numbers with addition (Koubarakis, 1994). Milliseconds will be used as the primary unit of time throughout this chapter, where, e.g., the time-point 4217 is be interpreted as "4.217 seconds after the beginning of the current reasoning problem".

Conceptually, the development of the world over a (possibly infinite) period of time can be viewed in two different ways: As a sequence of *states*, where each state provides a value to each feature (or "state variable") for a single common time-point, or as a set of *fluents*, where each fluent is a function of time specifying the value of a single feature at each time-point. The terms "feature" and "fluent" will sometimes be used interchangeably to refer to either a specific property of the world or the function specifying its value over time.

TAL Narratives. TAL is based on the use of narratives specifying background knowledge together with information about a specific reasoning problem. Narratives are initially specified in the narrative description language $\mathcal{L}$(ND), which provides support to a knowledge engineer through a set of high-level macros suitable for a particular task and may vary considerably between TAL logics. The semantics of the language is defined in terms of a translation into first- and second-order logical theories in the language $\mathcal{L}$(FL) which remains essentially unmodified.

The *narrative background specification* contains background knowledge associated with a reasoning domain. *Domain constraint statements* in $\mathcal{L}$(ND) represent facts true in all scenarios associated with a particular reasoning domain, such as the fact that the altitude of a helicopter will always be greater than zero. *Dependency constraint statements* can be used to represent causal theories or assertions which model intricate dependencies describing how and when features change relative to each other. *Action type specifications* specify knowledge about actions, including preconditions and context-dependent effects. Performing an action changes the state of the world according to a set of given rules, which are not necessarily deterministic. For example, the action of tossing a coin can be modeled within the TAL

framework, and there will be two possible result states.

The *narrative specification* contains information related to a specific problem instance or reasoning task. *Observation statements* represent observations made by an agent; in the context of planning, this may be used to specify information regarding the initial state. *Action occurrence statements* state which actions occur and provide parameters for those actions.

The Logical Base Language $\mathcal{L}$(FL). As noted above, TAL is order-sorted. An $\mathcal{L}$(FL) vocabulary specifies a number of sorts for values $\mathcal{V}_i$, each of which corresponds to a value domain. The sort $\mathcal{V}$ is assumed to be a supersort of all value sorts. There are also a number of sorts $\mathcal{F}_i$ for (reified) features, each one associated with a value sort $dom(\mathcal{F}_i) = \mathcal{V}_j$ for some j. The sort $\mathcal{F}$ is a supersort of all fluent sorts. Finally, there is a sort for actions $\mathcal{A}$ and a temporal sort $\mathcal{T}$.

Variables are typed and range over the values belonging to a specific sort. For convenience, they are usually given the same name as the sort but written in italics, possibly with a prime and/or an index. For example, the variables *box*, *box*′ and box_3 would be of the sort `box`. Similarly, variables named *t* are normally temporal variables, and variables named *n* are normally integer-valued.

$\mathcal{L}$(FL) uses three main predicates. The predicate $Holds : \mathcal{T} \times \mathcal{F} \times \mathcal{V}$ expresses the fact that a feature takes on a certain value at a certain time-point; for example, $Holds(0, \mathsf{attached}(\textbf{heli1}, \textbf{bx3}), \textbf{true})$ denotes the fact that $\mathsf{attached}(\textbf{heli1}, \textbf{bx3})$ has the value **true** at time 0. The predicate $Occlude : \mathcal{T} \times \mathcal{F}$ will be described in the discussion of persistence below. Finally, the predicate $Occurs : \mathcal{T} \times \mathcal{T} \times \mathcal{A}$ specifies what actions occur, and during what intervals of time. The equality predicate is also available, together with the $<$ and $\leq$ relations on the temporal sort $\mathcal{T}$. We sometimes write $\tau \leq \tau' < \tau''$ to denote the conjunction $\tau \leq \tau' \wedge \tau' < \tau''$, and similarly for other combinations of the relation symbols $\leq$ and $<$.

The function $value(\tau, f)$ returns the value of the fluent f at time τ. Formulas in $\mathcal{L}$(FL) are formed using these predicates and functions together with the standard connectives and quantifiers in the usual manner.

The High-Level Macro Language $\mathcal{L}$(ND). The following is a small subset of the $\mathcal{L}$(ND) language which is sufficient for the purpose of this chapter. Fluent formulas provide a convenient means of expressing complex conditions. Fixed fluent formulas provide a temporal context specifying when a fluent formula should hold.

Definition 7.3.1 (Fluent Formulas, Fixed Fluent Formulas) An *elementary fluent formula* has the form $f \hat{=} \omega$ where f is a fluent term of sort $\mathcal{F}_i$ and ω is a value term of sort $dom(\mathcal{F}_i)$. This formula denotes the atemporal fact that the feature f takes on the value ω. A *fluent formula* is an elementary fluent formula or a combination of fluent formulas formed with the standard logical connectives and quantification over values. A *fixed fluent formula* takes the form $[\tau, \tau']\ \alpha$, $(\tau, \tau']\ \alpha$, $[\tau, \tau')\ \alpha$, $(\tau, \tau')\ \alpha$, $[\tau, \infty)\ \alpha$, $(\tau, \infty)\ \alpha$ or $[\tau]\ \alpha$, where α is a fluent formula and τ and

τ' are temporal terms. This denotes the fact that the formula α holds at the given time-point or throughout the given temporal interval. □

The elementary fluent formula $f \triangleq$ **true** ($f \triangleq$ **false**) can be abbreviated f ($\neg f$). Note that $f = f'$ means that f and f' refer to the same feature, while $f \triangleq \omega$ denotes the fact that f takes on the value ω at the time-point specified by the temporal context. The infinity symbol ∞ is not a temporal term but denotes the lack of an upper bound; stating $[\tau, \infty)\ \phi$ is equivalent to stating $\forall t.t \geq \tau \rightarrow [t]\ \phi$.

Persistence and Action Effects. In most cases one would like to make the assumption that a feature is *persistent*, only changing values when there is a particular reason, such as an action whose effects explicitly modify the feature. This should be done in a non-monotonic and defeasible manner allowing the incremental addition of new reasons why the feature can change. Several of the fundamental problems in developing logics for reasoning about action and change are related to finding representationally and computationally efficient ways to encode such assumptions without the need to explicitly specify every feature that an action does *not* modify.

The TAL approach uses an occlusion predicate, where $Occlude(\tau, f)$ means that f is allowed to, but does not have to, change values at time τ. Action effects are specified in $\mathcal{L}$(ND) using the R and I reassignment macros. For example, $R([\tau]\ \alpha)$ models an action effect that causes α to hold at time τ, where α is an arbitrary fluent formula, while $I([\tau, \tau'])\ \alpha$ forces α to hold over an interval of time. This is translated into an $\mathcal{L}$(FL) formula where the *Holds* predicate is used to ensure that α holds at the specified time-point or interval and the *Occlude* predicate is used to ensure that all features occurring in α are occluded, also at the specified time-point or interval.

A circumscription axiom is used to ensure that features are only occluded when explicitly specified to be occluded. The resulting theory is then conjoined with axioms stating that at any time-point when a persistent feature is not occluded, it retains its value from the previous time-point (Doherty and Kvarnström, 2008; Doherty et al., 1998; Doherty, 1994).

Since persistence is not suitable for all features, TAL also supports *durational* features that revert to a default value when not occluded as well as *dynamic* fluents where no persistence or default value assumption is made. Feature types are specified in a fine-grained and flexible manner where the feature type can vary by instance or vary over time. For the remainder of this chapter, though, all features will be assumed to be persistent.

Reasoning about TAL narratives. In order to reason about a particular narrative in $\mathcal{L}$(ND), it is first mechanically translated into the base language $\mathcal{L}$(FL) using the *Trans* function as seen in Figure 7.7 on the next page. A circumscription policy is applied to the *Occurs* and *Occlude* predicates in the resulting theory, ensuring that no actions occur except those explicitly specified in action occurrence statements and no fluents are permitted to change except when explicitly specified. A set

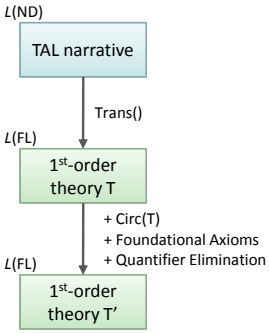


Figure 7.7: Reasoning in TAL.

of foundational axioms are added, including domain closure axioms and unique names axioms where appropriate. Finally, due to certain structural constraints on action type specifications and dependency constraint statements, quantifier elimination techniques can be used to reduce the resulting circumscribed second order theory to a first order theory (Doherty, Łukaszewicz, and Szałas, 1995, 1997; Doherty, 1994).

7.4 Planning for the UAV Domain

When developing the architecture for a system capable of autonomous action execution and goal achievement, one can envision a spectrum of possibilities ranging from each behavior and task being explicitly coded into the system, regardless of complexity, up to the other extreme where the system itself generates complex solutions composed from a set of very primitive low-level actions. With the former end of the spectrum generally leading to more computationally efficient solutions and the latter end generally being far more flexible in the event of new and potentially unexpected tasks being tackled, the proper choice is usually somewhere between the two extremes; in fact, several different points along the spectrum might be appropriate for use in different parts of a complex system. This is also the case for our UAV system, which provides a set of high-level actions such as "take off" and "fly to point A" but also makes use of planning techniques to compose such actions into plans satisfying a set of declaratively specified goals. The transportation of medical supplies is only one of many possible scenarios that can be modeled in this manner.

The planner used for this purpose is TALplanner (Doherty and Kvarnström, 1999, 2001; Kvarnström and Doherty, 2000; Kvarnström, 2005), a forward-chaining planner where planning domains and problem instances are specified using a version of TAL-C extended with new macros for plan operators, resource constraints, goal specifications, and other issues specific to the planning task. In addition to providing a declarative first-order semantics for planning domains, thereby serving

as a specification for the proper behavior of the planning algorithm, TAL is also used to specify a set of temporal control formulas acting as constraints on the set of valid plans. Such formulas can be used for specifying temporally extended goals that must be satisfied over the (predicted) execution of a plan. In addition, they can be used to constrain the forward-chaining search procedure, guiding the planner towards those parts of the search space that are more likely to contain plans of high quality in terms of flight time, delivery delays, or other quality measures.

In the remainder of this section, we will first show how TAL can be used for modeling the UAV logistics scenario (Section 7.4.1). We then discuss the use of control formulas in TALplanner and how they constrain the set of valid plans (Section 7.4.2). Due to the fine granularity with which operators have been modeled, the typical plan length for a small example with four boxes, one carrier, and one UAV is approximately 150 to 250 actions, depending on the initial state and the goal. Such plans can typically be generated in less than one second on a 1.8 GHz Pentium 4 machine. Most likely, some optimizations would be possible if necessary: The planning domain definition has currently been written for readability and ease of modification, not for performance.

See Kvarnström and Doherty (2000) or Kvarnström (2005) for further details regarding TALplanner.

7.4.1 Modeling the UAV Logistics Scenario in TAL

Though many traditional benchmark domains for planning are quite simple, and could be described and formalized in their entirety on a page or two, this is mainly due to two facts: First, many domains are designed to illustrate a specific point; second, a large proportion of the domains were defined at a time when planners could not be expected to handle more complex and detailed domain definitions due to limitations in computational capacity as well as in planning algorithms. The UAV domain discussed here, on the other hand, is intended to be used in a complex real world application where topics such as exact locations, distances and timing (as opposed to symbolic positions and unit timing for each action) are essential and cannot be abstracted away without severely compromising the average quality of a plan, or even the likelihood of it being executable at all. This means that the complete $\mathcal{L}$(ND) domain description with all associated operator specifications and control formulas is quite large, therefore a number of representative examples will be used.

In terms of value domains, we first require a set of domains to represent objects which have locations. Thus, the top level domain `locatable` has the subdomains `uav` and `carryable`, the latter of which has the subtypes `box` and `carrier`. Additionally, it may not be possible to place carriers at arbitrary positions due to terrain constraints, buildings, and other types of obstacles. For simplicity, the domain `carrier-position` represents intermediate locations where carriers may be placed when loading and unloading boxes. This may eventually be augmented with a method for querying the onboard GIS to dynamically find a suitable location for carrier placement; however, the requirement of predictability in a UAV deploy-

ment is likely to make the use of predefined carrier positions a legal necessity in many cases.

Each `locatable` object has a position which may vary over time. The x coordinate of a `locatable` is represented by the xpos feature, and the y coordinate by the ypos feature, taking values from a finite domain `fp` of fixed-point numbers (that is, numbers with a fixed number of decimals)[2]. The current altitude of each UAV is modeled using the altitude feature, also taking values from the domain `fp`. We appeal to the use of *semantic attachment* (Weyhrauch, 1980) techniques in the implementation of TAL and TALplanner by liberal use and invocation of built in mathematical functions and other functions associated with finite integer and fixed-point value domains.

Unlike some benchmark planning domains, where (for example) an unlimited number of boxes can be loaded into a single vehicle, our formalization of the UAV domain must comply with the physics of the real world. Consequently, each UAV has a limited carrying capacity, and each carrier has limited space available for boxes. Modeling this is not difficult, but a detailed model of issues involving weights and shapes would lead to unnecessary complexity in an area which is outside the focus of this chapter. Instead, a simpler model is used, where all boxes are assumed to be of similar shape, carrier capacities are modeled in terms of the number of boxes that will fit (carrier-capacity(*carrier*) : `fp`), and UAV capacities are modeled in terms of the number of boxes that it can carry (uav-capacity(*uav*) : `fp`).

It should also be mentioned that the UAV is only permitted to place boxes on specific areas on each carrier; boxes placed elsewhere could block the electromagnet from attaching to the carrier, or could fall off when the carrier is lifted. For this reason, the on-carrier(*box*, *carrier*) fluent is not boolean, but has three values representing the cases where the box is definitely not on the carrier, definitely correctly placed on the carrier, and perhaps blocking the carrier, respectively. Correctly deducing the third case also entails modeling the size of each carryable and the minimum safety distances between different types of carryables; this has been done but will not be further discussed here.

The boolean feature attached(*uav*, *carryable*) represents the fact that a certain UAV has attached its electromagnet to a certain carryable. Finally, a number of features are used for modeling abstract conditions such as whether the UAV is prepared and ready to fly, the most prominent one being state(*uav*) taking values from the domain `uavstate` = {**unprepared**, **ready-to-fly**, **ready-to-attach**, **ready-to-detach**}.

Operators

The UAV system provides a varied and complex set of functionalities that can be exposed to the planner as operators. Here, we will focus on the functionality that is the most relevant for the logistics missions used in this chapter: Flying to different

[2]Infinite domains are currently not allowed in TAL, and floating point numbers have a semantics which is considerably more difficult to formalize in logic. Since one can choose an arbitrary precision for a domain of fixed-point numbers, this is not a problem in practice.

locations, attaching and detaching carryable objects such as boxes and carriers, and (since we want a reasonably fine-grained model of the domain, in order to support fine-grained recovery) various forms of preparatory actions such as adjusting the altitude before attaching a box.

Even if the functionality exposed by the lower layers of the UAV architecture were seen as fixed and unalterable, there is still a great deal of flexibility in determining how this should be modeled in the planner: Even if there is a single parameterized *fly* functionality, for example, one may still choose to model this as several different operators for the purposes of planning if this has advantages in terms of readability. This is often the case for a planner such as TALplanner, which supports operator-specific control formulas that might only be applied in certain cases. To be more concrete, five different *fly* operators are used in the TAL domain specification created for the UAV domain: fly-empty-to-box, fly-empty-to-carrier, fly-box-to-carrier, fly-box-to-goal, and fly-carrier. Each of these operators has its own associated set of control formulas, because the action of flying without cargo to pick up a box is appropriate in different circumstances than flying with a box to place it on a carrier.

In addition to the flight operators, a UAV can also adjust-for-attach and then either attach-box or attach-carrier. After a subsequent climb-for-flying-with action, which reels in the winch and climbs to the standard flight altitude, it can fly the carryable to another location, adjust-for-detach, and either detach-box or detach-carrier depending on the type of carryable. After finishing with a climb-for-flying-empty action, the UAV is free to pursue other goals.

For all of these operators, there are a number of parameters, not all of which are relevant for the purpose of discussing the integration between planning and execution monitoring. A couple of examples are in order, though: climb-for-flying-empty(*uav*) requires a UAV as its only parameter. The operator fly-empty-to-carrier(*uav*, *fromx*,*fromy*, *carrier*, *tox*, *toy*) is somewhat more complex, requiring a UAV and its coordinates, plus a destination carrier and its coordinates, as parameters. (Note that if one models an area of 10000 meters square at a resolution of 1 centimeter, each coordinate has 10^6 possible values, and even with only a single UAV and a single carrier, the operator has 10^{24} ground instances. Obviously, TALplanner does not generate all ground instances of an operator, as some planners do.)

The exact effects and preconditions of these operators cannot be listed here in their entirety; again, they are not relevant for the purposes of the chapter. However, we will show one of the simplest operators in the syntax used by TALplanner together with its translation into $\mathcal{L}$(ND) and $\mathcal{L}$(FL).

```
#operator adjust-for-attach(uav, carryable, x, y)
  :at start, end
  :cost 500
  :timeconstraint 0 <= end - start <= 5 * UNITS_PER_SECOND
  :precond [start] near(uav, carryable, MAX_ATTACH_DIST)
                   & !hasCargo(uav)
                   & xpos(carryable) == x
                   & ypos(carryable) == y
  :effects [end] state(uav) := ready-to-attach
```

The operator is named adjust-for-attach and takes four parameters, two of which (x and y) are only used in operator-specific control formulas which are omitted here. The start and end time-points *start* and *end* can also be seen as implicit parameters and are instantiated by the planner as required during the search process. The :cost clause specifies a cost for this action, which is used if an optimizing search strategy is applied by the planner. Here, the cost is constant, but it can also depend on the arguments of the operator. The :timeconstraint clause provides a constraint on the timing of the operator, which may not be completely determined in advance. The :precondition clause makes use of several feature macros (features defined in terms of logic formulas), the meaning of which should be apparent from their usage. For example, hasCargo(*uav*) is defined to hold iff $\exists$*carryable*.attached(*uav*, *carryable*). The operator is applicable iff the UAV is sufficiently close to the carryable that should be attached and the UAV is not currently carrying any cargo; the conditions on x and y serve to bind these variables for use in control formulas. Finally, the only effect of this operator is that at the end, the state of the UAV is **ready-to-attach**.

The TAL-C action type specification corresponding to this would be defined as follows. Cost is omitted since this concept is only used during the planning phase and is not part of TAL. Free variables are assumed to be universally quantified.

$$
\begin{aligned}
&[\mathit{start}, \mathit{end}]\ \text{adjust-for-attach}(\mathit{uav}, \mathit{carryable}, x, y) \rightarrow \\
&0 \leq \mathit{end} - \mathit{start} \leq 5 \cdot \mathbf{UNITS\_PER\_SECOND} \wedge \\
&([\mathit{start}]\ \text{near}(\mathit{uav}, \mathit{carryable}, \mathbf{MAX\_ATTACH\_DIST}) \wedge \\
&\neg\text{hasCargo}(\mathit{uav}) \wedge \text{xpos}(\mathit{carryable}) \mathrel{\hat{=}} x \wedge \text{ypos}(\mathit{carryable}) \mathrel{\hat{=}} y \rightarrow \\
&R([\mathit{end}]\ \text{state}(\mathit{uav}) \mathrel{\hat{=}} \mathbf{ready\text{-}to\text{-}attach}))
\end{aligned}
$$

This would be translated into the following $\mathcal{L}$(FL) action type specification:

$$
\begin{aligned}
&\mathit{Occurs}(\mathit{start}, \mathit{end}, \text{adjust-for-attach}(\mathit{uav}, \mathit{carryable}, x, y)) \rightarrow \\
&0 \leq \mathit{end} - \mathit{start} \leq 5 \cdot \mathbf{UNITS\_PER\_SECOND} \wedge \\
&(\mathit{Holds}(\mathit{start}, \text{near}(\mathit{uav}, \mathit{carryable}, \mathbf{MAX\_ATTACH\_DIST}), \mathbf{true}) \wedge \\
&\neg\mathit{Holds}(\mathit{start}, \text{hasCargo}(\mathit{uav}), \mathbf{true}) \wedge \\
&\mathit{Holds}(\mathit{start}, \text{xpos}(\mathit{carryable}), x) \wedge \mathit{Holds}(\mathit{start}, \text{ypos}(\mathit{carryable}), y) \rightarrow \\
&\mathit{Holds}(\mathit{end}, \text{state}(\mathit{uav}), \mathbf{ready\text{-}to\text{-}attach}) \wedge \\
&\mathit{Occlude}(\mathit{end}, \text{state}(\mathit{uav})))
\end{aligned}
$$

7.4.2 Control Formulas in TALplanner

Given the operators described above together with an initial state and a set of formulas that must be satisfied in any goal state, the task of the planner is to search for a valid and executable plan that ends in a goal state. If we (for the sake of discussion) temporarily restrict ourselves to finding sequential plans, applying a forward-chaining algorithm to this task entails a search space where the root node is the empty action sequence and where the children of any node n are exactly those generated by appending one more applicable action to the end of the action sequence associated with n.

Clearly, each node n associated with an action sequence p can also be viewed as corresponding to a finite state sequence $[s_0, s_1, \ldots, s_m]$ – the sequence that would be generated by executing p starting in the initial state. This state sequence, in

turn, can be seen as corresponding to a TAL interpretation. This leads us directly to the possibility of specifying constraints on a plan in terms of TAL formulas to be satisfied by the corresponding interpretation.

Before going into further detail, a small example will be presented.

Example 7.4.1 (Global Control Formula) In the UAV domain, a box should only be moved if the goal requires it to be elsewhere, or if it is blocking some other action. The first condition holds if the box is farther away from its ideal goal coordinates than a given (possibly context-specific) threshold. One cannot require the box to be *exactly* at a given set of coordinates: If a failure is detected the system may need to replan, and one does not want to move boxes again merely because they may be centimeters away from an ideal goal location. The second condition might hold if the box is too close to a carrier position according to a specified safety margin. These conditions are simplified and modularized using several feature macros defined in terms of basic features: close-enough-to-goal(box), need-to-move-temporarily(box), and is-at($locatable, x, y$).

$$\forall t, box, x, y.$$
$$[t]\ \text{is-at}(box, x, y) \rightarrow [t+1]\ \text{is-at}(box, x, y) \vee$$
$$[t]\ \neg\text{close-enough-to-goal}(box) \vee$$
$$[t]\ \text{need-to-move-temporarily}(box) \qquad \square$$

Some control formulas are always satisfied by inaction; for example, the empty plan generates a state sequence where no boxes are moved, which will satisfy the formula above. Formulas may also require certain changes to take place, however. For example, one could state that if a UAV takes off, it must begin flying to a destination within 30 seconds; otherwise it is wasting fuel and might as well have waited on the ground.

If fluents are assumed to be persistent (not change values) after the final action in a plan, then a plan consisting of a single takeoff action will permit the conclusion that the UAV takes off and then remains stationary indefinitely, which violates the control formula. This is obviously wrong, since the only reason why the UAV remains stationary is that the plan search algorithm has not yet added a flight action. Only after the planner has actually added actions predicted to take 30 seconds or more, without adding a flight action, should the control formula be considered to be violated.

This is achieved by considering the state sequence $[s_0, s_1, \ldots, s_m]$ associated with an intermediate node in the search tree to be a *partial* state sequence, a prefix of the final sequence that will eventually be generated. Similarly, the TAL interpretation corresponding to an intermediate node is viewed as a partial interpretation $\mathcal{I}$, where fluent values up to the time of state s_m are completely determined, after which they are assumed to be completely unknown. If ϕ is a control formula and $\mathcal{I} \models \neg\phi$, then both this node and any descendant must necessarily violate the control formula (because descendant nodes can only add new information after s_m which does not retract any previous conclusions), and the planner can reject the node and backtrack. How to test this efficiently is discussed in Kvarnström (2002; 2005).

In addition to global control formulas, TALplanner also permits the use of operator-specific control. Such formulas are similar to preconditions, but whereas preconditions are intended to model constraints on when it is "physically" possible to use an operator, operator-specific control is intended to model constraints on when using the operator is recommended.

Example 7.4.2 (Operator-specific Control) Suppose one wants to express the constraint that a UAV should not prepare for attaching to a carrier where there are potentially misplaced boxes, because such a carrier may not be correctly balanced. This control formula can be declared locally in the adjust-for-attach operator, which also gives it access to operator parameters. Specifically, the *carryable* parameter indicates the relevant carryable, which may or may not be a carrier. This leads to the following conditionalized control formula, which is violated by any action preparing to attach to a carrier where there is a box which is closer than the designated safety distance and is not correctly placed:

$[start]\ \forall carrier, box.$
$carrier = carryable \wedge \mathsf{near}(box, carrier, \mathsf{safetyDistance}(box, carrier)) \rightarrow$
$\mathsf{on\text{-}carrier}(box, carrier) \hat{=}$ **correctly_placed** □

7.5 Execution Monitoring

Classical planners are built on the fundamental assumption that the only agent causing change in the environment is the planner itself, or rather, the system or systems that will eventually execute the plan that it generates. Furthermore, they assume that all information provided to the planner as part of the initial state and the operator specifications is accurate. Though this may in some cases be a reasonable approximation of reality, it is more often manifestly untrue: Numerous other agents might manipulate the environment of an autonomous system in ways that may prevent the successful execution of a plan, and actions can sometimes fail to have the effects that were modeled in a planning domain specification regardless of the effort spent modeling all possible contingencies. Consequently, robust performance in a noisy environment requires some form of supervision, where the execution of a plan is constantly monitored in order to detect any discrepancies and recover from potential or actual failures. For example, a UAV might accidentally drop its cargo; thus, it must monitor the condition that if a box is attached, it must remain attached until the UAV reaches its intended destination. This is an example of a *safety constraint*, a condition that must be maintained during the execution of an action or across the execution of multiple actions. The carrier can also be too heavy, which means that one must be able to detect takeoff failures where the UAV fails to gain sufficient altitude. This can be called a *progress constraint*: Instead of maintaining a condition, a condition must be achieved within a certain period of time.

The requirement for monitoring leads to the question of what conditions should be monitored, and how such conditions should be specified. Clearly, there are

certain contingencies that would best be monitored by the low-level implementation of an operation or behavior, but universal use of this principle would lead to excessively complex action implementations with duplication of failure detection functionalities and a lack of modularity. As an alternative, the monitoring of failures and the recovery from unintentional situations could be separated from the execution of actions and plans and lifted into a higher level *execution monitor* (Ben Lamine and Kabanza, 2002; Bjäreland, 2001; De Giacomo, Reiter, and Soutchanski, 1998; Fernández and Simmons, 1998; Fichtner, Grossmann, and Thielscher, 2003; Gat et al., 1990), where the constraints to be monitored should preferably be specified in an expressive declarative formalism. If a constraint is violated, the execution system should be signaled, after which the UAV can react and attempt to recover from the failure. This is the approach taken in this thesis.

Our system for monitoring the correct execution of a plan is based on an intuition similar to that underlying the temporal control formulas used in TALplanner. As a plan is being executed, information about the surrounding environment is sampled at a given frequency. Each new sampling point generates a new state which provides information about all fluents used by the current monitor formulas, thereby providing information about the *actual* state of the world at that particular point in time, as opposed to what could be *predicted* from the domain specification. Concatenating all states generated by this sampling process yields a state sequence that corresponds to a partial logical interpretation, where "past" and "present" states are completely specified whereas "future" states are completely undefined (Section 7.8 on page 117).

Given that both actual and predicted states are available, one obvious approach to monitoring would be to simply compare these states and signal a violation as soon as a discrepancy is found. Unfortunately, the trivial application of this approach is not sufficient, because not all discrepancies are fatal: If the altitude was predicted to be 5 meters and the current measurement turns out to be 4.984 meters, then one most likely does not have to abort the mission. Additionally, some information about the environment might be expensive or difficult to sense, in which case the operator or domain designer should be given more control over when and where such information is used, rather than forcing the system to gather this information continuously in order to provide sufficient information for state generation. Finally, the richer the domain model is, the more the planner can predict about the development of the world; this should not necessarily lead to all those conditions being monitored, if they are not relevant to the correct execution of a plan. Determining which of these predictions are truly important for goal achievement and which are less relevant, and weighing this importance against the difficulty or expense involved in sensing, is best done by a human.

For these reasons, most conditions to be monitored are explicitly specified using a variation of Temporal Action Logic (Section 7.5.1), though many conditions *can* be automatically generated within the same framework if so desired (Section 7.7 on page 115). For example, a very simple monitor formula might monitor the constraint that at any time-point, $\forall uav.\mathsf{altitude}(uav) \leq 50$. Through the use of logic, conditions are given an expressive formal declarative semantics where both

safety and progress conditions can be defined and monitored.

As each new sensed state arrives into the system, the current set of monitor formulas is tested against the incrementally constructed partial logical model using a progression algorithm (Section 7.5.2 on page 110), and any violation is reported to the execution system which can take the appropriate action (Section 7.5.3 on page 112).

7.5.1 Execution Monitor Formulas

Having decided that the conditions to be tested by the execution monitor should be specified in the form of logic formulas, we still retain a great deal of freedom in choosing exactly which logic should be used. However, there are several important considerations that can be used for guidance.

First, the logic must be able to express conditions over time. In restricted cases, the conditions to be tested could consist of simple state constraints specifying requirements that must hold in each individual state throughout the execution of a plan. In the general case, though, one would need the ability to specify conditions *across* states, such as the fact that when a UAV has attached to a box, the box must remain attached until released.

Second, the logic should be able to express metric constraints on time, in order to provide support for conditions such as the UAV succeeding in attaching a box within at most 10 seconds.

These are the most important constraints on expressivity, and had these been the only constraints on our system, it would have been possible to use an unmodified version of TAL-C to express monitor formulas. After all, TAL-C has been used to great success in specifying control formulas for the planning phase, and has support for temporally extended constraints as well as metric time. However, there is a third and more pragmatic constraint that must also be satisfied: It must be possible to test each monitor formula *incrementally* and *efficiently* as each new state arrives from DyKnow into the execution monitor, because violations must be detected as early as possible, and there may be a large number of monitor formulas to be tested using the limited computational capabilities of the onboard computers on a UAV.

For TALplanner, the main strategy for verifying that control formulas are not violated is based on the use of plain formula evaluation in a partial interpretation. The efficiency of this strategy is predicated on the use of an extensive pre-planning analysis phase where the control formulas specified for a given domain are analyzed relative to the operators available to the planner (Kvarnström, 2002). For the execution monitoring system, the prerequisites for the formula analysis phase are not satisfied: The very reason for the existence of the system is the potential for unforeseen and unpredictable events and failures, rendering the analysis relative to specific operators specifying "what could happen" ineffective. For this reason, a formula progression algorithm would be more appropriate for this application, and such algorithms are more easily applied to formulas in a syntax such as that used in modal tense logics such as LTL (Linear Temporal Logic (Emerson, 1990)) and

MITL (Metric Interval Temporal Logic (Alur and Henzinger, 1992; Alur, Feder, and Henzinger, 1991)).

Note again the wording above: Progression is more easily applied to formulas in a *syntax* such as that used in modal tense logics. In fact, there is no requirement for actually using such a logic; instead, it is possible to create a new variation of the $\mathcal{L}$(ND) surface language for TAL-C containing operators similar to those in MITL together with a translation into the same base logic $\mathcal{L}$(FL). Doing this has the clear advantage of providing a common semantic ground for planning and execution monitoring, regardless of the surface syntax of a formula.[3] It should be noted that the ability to adapt the surface language to specific applications is in fact one of the main reasons behind the use of two different languages in TAL; the $\mathcal{L}$(ND) language has already been adapted in different ways for planning (Kvarnström and Doherty, 2000; Kvarnström, 2005), object-oriented modeling (Gustafsson and Kvarnström, 2004), and other tasks.

Monitor Formulas in TAL

Three new *tense operators* have been introduced into $\mathcal{L}$(ND) for use in formula progression: U (until), $\Diamond$ (eventually), and $\Box$ (always). Note that like all expressions in $\mathcal{L}$(ND), these operators are macros on top of the first order base language $\mathcal{L}$(FL). We also introduce the concept of a *monitor formula* in TAL.

Definition 7.5.1 (Monitor Formula) A *monitor formula* is one of the following:

- $\tau \leq \tau'$, $\tau < \tau'$, or $\tau = \tau'$, where τ and τ' are temporal terms,
- $\omega = \omega'$, where ω and ω' are value terms,
- a fluent formula,
- $\phi \,\mathsf{U}_{[\tau,\tau']}\, \psi$, where ϕ and ψ are monitor formulas and τ and τ' are temporal terms,
- $\Diamond_{[\tau,\tau']}\, \phi$, where ϕ is a monitor formula and τ and τ' are temporal terms,
- $\Box_{[\tau,\tau']}\, \phi$, where ϕ is a monitor formula and τ and τ' are temporal terms, and
- a combination of monitor formulas using the standard logical connectives and quantification over values.

The shorthand notation $\phi \,\mathsf{U}\, \psi \equiv \phi \,\mathsf{U}_{[0,\infty)}\, \psi$, $\Diamond \phi \equiv \Diamond_{[0,\infty)}\, \phi$, and $\Box \phi \equiv \Box_{[0,\infty)}\, \phi$ is also permitted in monitor formulas. □

Whereas other logic formulas in $\mathcal{L}$(ND) use absolute time (as in the fixed fluent formula $[\tau]\ f \,\hat{=}\, v$), monitor formulas use relative time, where each formula is evaluated relative to a "current" time-point. The semantics of these formulas will be defined in terms of a translation into $\mathcal{L}$(FL) satisfying the following conditions:

[3] The use of tense operators in TAL was in fact first introduced in TALplanner, which provides both the standard TAL syntax and a tense logic syntax for its control formulas.

- The formula $\phi \, \mathsf{U}_{[\tau,\tau']} \, \psi$ ("until") holds at time t iff ψ holds at some state with time $t' \in [t+\tau, t+\tau']$ and ϕ holds until then (at all states in $[t, t')$, which may be an empty interval).
- The formula $\Diamond_{[\tau,\tau']} \, \phi$ ("eventually") is equivalent to $\mathtt{true} \, \mathsf{U}_{[\tau,\tau']} \, \phi$ and holds at t iff ϕ holds in some state with time $t' \in [t+\tau, t+\tau']$.
- The formula $\Box_{[\tau,\tau']} \, \phi$ is equivalent to $\neg \Diamond_{[\tau,\tau']} \neg\phi$ and holds at t iff ϕ holds in all states with time $t' \in [t+\tau, t+\tau']$.

Definition 7.5.2 (Translation of Monitor Formulas) Let $\overline{\tau}$ be a temporal term and γ be a monitor formula intended to be evaluated at $\overline{\tau}$. Then, the following procedure returns an equivalent formula in $\mathcal{L}$(ND) without tense operators.

$$
\begin{array}{lll}
\mathsf{TransMonitor}(\overline{\tau}, Qx.\phi) & \stackrel{\text{def}}{=} & Qx.\mathsf{TransMonitor}(\overline{\tau}, \phi), \text{ where } Q \text{ is a quantifier} \\
\mathsf{TransMonitor}(\overline{\tau}, \phi \otimes \psi) & \stackrel{\text{def}}{=} & \mathsf{TransMonitor}(\overline{\tau}, \phi) \otimes \mathsf{TransMonitor}(\overline{\tau}, \psi), \text{ where } \otimes \text{ is} \\
 & & \quad \text{a binary connective} \\
\mathsf{TransMonitor}(\overline{\tau}, \neg\phi) & \stackrel{\text{def}}{=} & \neg\mathsf{TransMonitor}(\overline{\tau}, \phi) \\
\mathsf{TransMonitor}(\overline{\tau}, f \mathrel{\hat{=}} v) & \stackrel{\text{def}}{=} & [\overline{\tau}] \; f \mathrel{\hat{=}} v \\
\mathsf{TransMonitor}(\overline{\tau}, \gamma) & \stackrel{\text{def}}{=} & \gamma, \text{ where } \gamma \text{ has no tense operators} \\
\mathsf{TransMonitor}(\overline{\tau}, \phi \, \mathsf{U}_{[\tau,\tau']} \, \psi) & \stackrel{\text{def}}{=} & \exists t[\overline{\tau} + \tau \leq t \leq \overline{\tau} + \tau' \wedge \\
 & & \quad \mathsf{TransMonitor}(t, \psi) \wedge \forall t'[\overline{\tau} \leq t' < t \rightarrow \mathsf{TransMonitor}(t', \phi)]] \\
\mathsf{TransMonitor}(\overline{\tau}, \Box_{[\tau,\tau']} \, \phi) & \stackrel{\text{def}}{=} & \forall t[\overline{\tau} + \tau \leq t \leq \overline{\tau} + \tau' \rightarrow \mathsf{TransMonitor}(t, \phi)] \\
\mathsf{TransMonitor}(\overline{\tau}, \Diamond_{[\tau,\tau']} \, \phi) & \stackrel{\text{def}}{=} & \exists t[\overline{\tau} + \tau \leq t \leq \overline{\tau} + \tau' \wedge \mathsf{TransMonitor}(t, \phi)]
\end{array}
$$

Line 5 handles elementary fluent formulas, which are not full logic formulas in themselves and require the addition of an explicit temporal context $[\overline{\tau}]$. Other formulas without occurrences of tense operators, for example value comparisons of the form $v = w$, require no temporal context and are handled in line 6.

The *Trans* translation function is extended for tense monitor formulas by defining $\mathit{Trans}(\gamma) \stackrel{\text{def}}{=} \mathit{Trans}(\mathsf{TransMonitor}(0, \gamma))$. □

A few examples may be in order.

Example 7.5.1 Suppose that whenever a UAV is moving, the winch should not be lowered. In this example, "moving" should not be interpreted as "having a speed not identical to zero", since the UAV might not be able to stay perfectly still when hovering and sensor noise may cause the sensed speed to fluctuate slightly. Instead, the UAV is considered to be still when its sensed speed is at most s_{min}. We assume the existence of a winch feature representing how far the winch has been extended and a limit w_{min} determining when the winch is considered to be lowered, which leads to the following monitor formula.

$\Box \, \forall uav.\mathsf{speed}(uav) > s_{\mathsf{min}} \rightarrow \mathsf{winch}(uav) \leq w_{\mathsf{min}}$ □

Note that this does not in itself *cause* the UAV to behave in the desired manner. This has to be achieved in the lower level implementations of the helicopter control software. This monitor formula instead serves as a method for detecting the failure of the helicopter control software to function according to specifications.

Example 7.5.2 Suppose that a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification:

$$\Box \forall uav.(\mathsf{power}(uav) > M \rightarrow \mathsf{power} < f \cdot M \, \mathsf{U}_{[0,\tau]} \, \Box_{[0,\tau']} \, \mathsf{power}(uav) \leq M)$$ □

Further examples will be shown in Section 7.6, after the introduction of operator-specific monitor formulas and a means for formulas to explicitly represent queries about aspects of the execution state of the autonomous system.

7.5.2 Checking Monitor Conditions using Formula Progression

We now have a syntax and a semantics for conditions to be monitored during execution. Given the complete state sequence corresponding to the events taking place during the execution of a plan, a straight-forward implementation of the semantics can be used to test whether a monitor formula is violated. This is sufficient for post-execution analysis, but true execution monitoring requires prompt detection of potential or actual failures *during* execution.

A formula progression algorithm can be used for this purpose (Bacchus and Kabanza, 1996, 1998). By definition, a formula ϕ holds in the state sequence $[s_0, s_1, \ldots, s_n]$ iff $\mathsf{Progress}(\phi, s_0)$ holds in $[s_1, \ldots, s_n]$. Thus, a monitor formula can be incrementally progressed through each new state that arrives from DyKnow, evaluating only those parts of the formula that refer to the newly received state.

As soon as sufficient information has been received to determine that the monitor formula must be violated regardless of the future development of the world, the formula $\perp$ (false) is returned. For example, this will happen as soon as the formula $\Box\, \mathsf{speed} < 50$ is progressed through a state where $\mathsf{speed} \geq 50$. Using progression thus ensures that failures are detected quickly and without evaluating formulas in the same state more than once.

The result of progression might also be $\top$ (true), in which case the formula must hold regardless of what happens "in the future". This will occur if the formula is of the form $\Diamond \phi$ (eventually, ϕ will hold), and one has reached a state where ϕ indeed does hold. In other cases, the state sequence will comply with the constraint "so far", and progression will return a new and potentially modified formula that should be progressed again as soon as another state is available.

Since states are not first-class objects in TAL, the state-based definition of progression must be altered slightly. Instead of taking a state as an argument, the procedure below is provided with an interpretation together with the time-point corresponding to the state through which the formula should be progressed.

An additional change improves performance and flexibility for the situation where a single sample period corresponds to multiple discrete TAL time-points. Specifically, if samples are known to arrive every m time-points and one is progressing a formula ϕ where the lower temporal bounds of all tense operators are multiples of m', it is possible to progress ϕ through $n = \gcd(m, m')$ time-points in a

single step. The value of n is assumed to be provided to the progression procedure as defined below. This permits the temporal unit to be completely decoupled from the sample rate while at the same time retaining the standard TAL-based semantics, where states exist at every discrete time-point.

For example, suppose a TAL time-point corresponds to 1 ms. If samples arrive every 50 ms, the formula $\Diamond_{[0,3037]}\ \phi$ can be progressed through $\gcd(50, 0) = 50$ time-points in a single step, while the formula $\Diamond_{[40,3037]}\ \phi$ can be progressed through $\gcd(50, 40) = 10$ time-points. If samples arrive every 100 ms, the formula $\Diamond_{[0,3037]}\ \phi$ can be progressed through $\gcd(100, 0) = 100$ time-points in a single step, while the formula $\Diamond_{[40,3037]}\ \phi$ can be progressed through $\gcd(100, 40) = 20$ time-points. Thus, formula definitions and sample rates can be altered independently (which would not be the case if a TAL time-point was defined to be a single sample interval), and progression automatically adapts to the current situation.

The progression algorithm below satisfies the following property. Let n be a progression step measured in time-points, ϕ be a monitor formula where all times are multiples of n, τ a numeric time-point, and $\mathcal{I}$ a TAL interpretation. Then, $\mathsf{Progress}(\phi, \tau, n, \mathcal{I})$ will hold at $\tau + n$ in $\mathcal{I}$ iff ϕ holds at τ in $\mathcal{I}$. More formally,

$$\mathcal{I} \models \mathit{Trans}(\mathsf{TransMonitor}(\tau, \phi)) \text{ iff } \mathcal{I} \models \mathit{Trans}(\mathsf{TransMonitor}(\tau{+}n, \mathsf{Progress}(\phi, \tau, n, \mathcal{I}))),$$

where $\mathsf{TransMonitor}(\tau, \phi)$ is the translation of a monitor formula ϕ into $\mathcal{L}(\mathrm{FL})$ relative to the time-point τ as described above.

Definition 7.5.3 (Progression of Monitor Formulas) The following algorithm is used for progression of monitor formulas. Special cases for $\Box$ and $\Diamond$ can also be introduced for performance.

procedure $\mathsf{Progress}(\phi, \tau, n, \mathcal{I})$
if $\phi = f(\overline{x}) \mathrel{\hat{=}} v$
  if $\mathcal{I} \models \mathit{Trans}([\tau]\ \phi)$ **return** $\top$ **else return** $\bot$
if $\phi = \neg\phi_1$ **return** $\neg\mathsf{Progress}(\phi_1, \tau, n, \mathcal{I})$
if $\phi = \phi_1 \otimes \phi_2$ **return** $\mathsf{Progress}(\phi_1, \tau, n, \mathcal{I}) \otimes \mathsf{Progress}(\phi_2, \tau, n, \mathcal{I})$
if $\phi = \forall x.\phi$ // where x belongs to the finite domain X
  return $\bigwedge_{c \in X} \mathsf{Progress}(\phi[x \mapsto c], \tau, n, \mathcal{I})$
if $\phi = \exists x.\phi$ // where x belongs to the finite domain X
  return $\bigvee_{c \in X} \mathsf{Progress}(\phi[x \mapsto c], \tau, n, \mathcal{I})$
if ϕ contains no tense operator
  if $\mathcal{I} \models \mathit{Trans}(\phi)$ **return** $\top$ **else return** $\bot$
if $\phi = \phi_1\ \mathsf{U}_{[\tau_1,\tau_2]}\ \phi_2$
  if $\tau_2 < 0$ **return** $\bot$
  elsif $0 \in [\tau_1, \tau_2]$ **return** $\mathsf{Progress}(\phi_2, \tau, n, \mathcal{I}) \vee$
    $(\mathsf{Progress}(\phi_1, \tau, n, \mathcal{I}) \wedge (\phi_1\ \mathsf{U}_{[\tau_1-n,\tau_2-n]}\ \phi_2))$
  else return $\mathsf{Progress}(\phi_1, \tau, n, \mathcal{I}) \wedge (\phi_1\ \mathsf{U}_{[\tau_1-n,\tau_2-n]}\ \phi_2)$

The result of Progress is simplified using the rules $\neg\bot = \top$, $(\bot \wedge \alpha) = (\alpha \wedge \bot) = \bot$, $(\bot \vee \alpha) = (\alpha \vee \bot) = \alpha$, $\neg\top = \bot$, $(\top \wedge \alpha) = (\alpha \wedge \top) = \alpha$, and $(\top \vee \alpha) = (\alpha \vee \top) = \top$. Further simplification is possible using identities such as $\Diamond_{[0,\tau]}\phi \wedge \Diamond_{[0,\tau']}\phi \equiv \Diamond_{[0,\min(\tau,\tau'')]}\phi$. □

7.5.3 Recovery from Failures

Any monitor formula violation signals a potential or actual failure from which the system must attempt to recover in order to achieve its designated goals.

Recovery is a complex topic, especially when combined with the stringent safety regulations associated with flying an autonomous unmanned vehicle and the possibility of having time-dependent goals as well as time-dependent constraints on the behavior of the vehicle. For example, a UAV might only be allowed to fly in certain areas at certain times. There may also be complex temporal dependencies between operations intended to be carried out by different UAVs, and given that one UAV has failed, optimal or near-optimal behavior for the aggregated system might require further modifications to the plan of another UAV. For example, if **heli1** fails to deliver a box of medicine on time, **heli2** might have to be rerouted in order to meet a goal deadline. For these reasons, our first iteration of the recovery system has not tackled incremental plan repair and modifications, even though such properties may be desirable in the long run. Instead, recovery is mainly performed through replanning, where a single planning domain specification and planning algorithm can be used for both the initial planning phase and the recovery phase. Given that the planner is sufficiently fast when generating new plans, this does not adversely affect the execution of a fully autonomous mission.

Thus, having detected a failure, the first action of a UAV is to cancel the current plan, execute an emergency break if required, and then go into autonomous hover mode. Currently, we take advantage of the fact that our UAV is rotor-based and can hover. For fixed-wing platforms, this is not an option and one would have to go into a loiter mode if the recovery involves time-consuming computation.

This is followed by the execution of a *recovery operator*, if one is associated with the violated monitor formula. The recovery operator can serve two purposes: It can specify emergency recovery procedures that must be initiated immediately without waiting for replanning, and it can permit the execution system to adjust its assumptions about what can and cannot be done. For example, if a UAV fails to take off with a certain carrier, it may have to adjust its assumptions about how many boxes it is able to lift (or, equivalently, how heavy the boxes on the carrier are). The associated recovery operator can perform this adjustment, feeding back information from the failure into the information given to the planner for replanning. The implementation of a recovery operator can also detect the fact that the UAV has attempted and failed to recover from the same fault too many times and choose whether to give up, try another method, remove some goals in order to succeed with the remaining goals, or contact a human for further guidance.

After executing a recovery operator, the UAV must find sufficient information about its environment to construct a new initial state to be given to the planner. In the simulated execution system, this information can be extracted by DyKnow directly from the simulator; even though a box may have bounced and rolled after having been dropped, the simulator will obviously know exactly where it ended up. In the real world, locating and identifying objects is a more complex topic which has not yet been tested in the context of execution monitoring and replanning. Possibly, the next iteration of the recovery system will include some de-

gree of intervention by a human operator in order to locate objects, or confirm a UAV's hypotheses regarding object locations, before we move on to completely autonomous recovery. For some ideas of how the topic of object identification can be approached see Chapter 8.

Having constructed a formal representation of the current state, the plan executor can once again call the planner and replan from this state. This yields a new plan, which must take into account the new situation as well as any time-related constraints.

7.6 Further Integration of Planning and Monitoring

Making full use of the execution monitoring system developed in the previous section requires a higher degree of integration between the planning phase and the execution monitoring phase. The following example illustrates some of the difficulties associated with doing execution monitoring without such integration.

Example 7.6.1 Whenever a UAV attaches to a carryable, it should remain attached until explicitly detached. If only global monitor formulas are available, this condition must be approximated using knowledge about the behavior of the UAV when it is in flight and when it attaches or detaches a carryable. Suppose that UAVs always descend below 4 meters to attach a carryable, always stay (considerably) above an altitude of 7 meters while in flight, and always descend to below 4 meters before detaching a carryable. The following condition states that whenever a UAV is attached to a carryable and has achieved the proper flight altitude, it must remain attached to the carryable until it is at the proper altitude for detaching it:

$$\Box\, \forall uav, carryable.\ \mathsf{attached}(uav, carryable) \wedge \mathsf{altitude}(uav) \geq 7.0 \rightarrow \mathsf{attached}(uav, carryable)\ \mathsf{U}\ \mathsf{altitude}(uav) \leq 4.0$$

This formula is inefficient, however: It will be tested in all states and for all UAVs and boxes, regardless of whether an attach action has been performed. It is also brittle: If the UAV drops the carrier before reaching flight altitude, no failure will be signaled, because the formula only triggers once the UAV rises above an altitude of 7 meters. If the margin between the two altitudes is decreased, there will be a smaller interval in which a carryable might be dropped without detection, but only at the cost of increasing the risk that the formula is triggered during a proper detach action. As soon as the UAV descends to below 4 meters for any reason, the monitor will cease testing whether the carryable remains attached, even if the descent was temporary and not intended to lead to a detach action. □

In the remainder of this section, we will discuss how monitor formulas can be associated with individual operators and provided to the execution system as part of the plan, improving the flexibility and modularity of the system as well as the expressive power of the formulas (Section 7.6.1). We will also discuss how to specify monitoring formulas of different longevity, either terminating at the end of

an action or continuing to monitor conditions across an arbitrary interval of time (Section 7.6.2).

7.6.1 Operator-Specific Monitor Formulas

The first step in a deeper integration between planning and execution monitoring involves allowing execution monitor formulas to be explicitly associated with specific operator types. Unlike global monitor formulas, such formulas are not activated before plan execution but before the execution of a particular *step* in the plan, which provides the ability to contextualize a monitor condition relative to a particular action. An operator-specific monitor formula can also directly refer to the arguments of the associated operator. As for global formulas, a recovery action can be associated with each formula.

We are not yet ready to provide an improved formula for the motivational example above. However, to illustrate the principle, consider the following example.

Example 7.6.2 When a UAV attempts to attach to a box, the attempt may fail; therefore, the success of the action should be monitored. The attach-box operator takes four arguments: A *uav*, a *box*, and the *x* and *y* coordinates of the box. Making use of the first two arguments, the following operator-specific monitor formula may be suitable:

$\Diamond_{[0,5000]} \Box_{[0,1000]}$ attached(*uav*, *box*)

Within 5000 ms, the box should be attached to the UAV, and it should remain attached for at least 1000 ms. The latter condition is intended to protect against problems during the attachment phase, where the electromagnet might attach to the box during a very short period of time even though the ultimate result is failure. □

When a plan is generated by TALplanner, each action in the plan is annotated with a set of instantiated operator-specific monitor formulas. Continuing the previous example, the action [120000, 125000] attach-box(**heli1**, **bx7**, **127.52**, **5821.23**) would be annotated with the instantiated formula $\Diamond_{[0,5000]} \Box_{[0,1000]}$ attached(**heli1**, **bx7**). During execution, this instantiated formula is added to the execution monitor immediately before beginning execution of the attach-box action.

7.6.2 Execution Flags

The power of monitor formulas can be extended further by also giving access to certain information about the plan execution state, in addition to the world state. In the motivational example above, one would like to state that once a carrier has been attached to the UAV, it should remain attached until the UAV intentionally detaches it, that is, *until the corresponding detach action is executed*. One may also want to state that a certain fact should *hold during* the execution of an action, or that an effect should be *achieved during* the execution of an action.

In order to allow monitor formulas to query the execution state of the agent, we introduce the use of *execution flags*. An execution flag is a standard parameterized boolean fluent which holds exactly when the corresponding operator is being

executed with a specific set of arguments. By convention, this fluent will generally be named by prepending "executing-" to the name of the corresponding operator. For example, the attach-box action is associated with the executing-attach-box execution flag, which takes a subset of the operator's parameters. The execution subsystem is responsible for setting this flag when execution starts and clearing it when execution ends.

Example 7.6.3 Consider the climb-for-flying-empty(uav) operator, which should cause the UAV to ascend to its designated flight altitude. Here, one may wish to monitor the fact that the UAV truly ends up at its flight altitude. This can be achieved using the formula executing-climb-for-flying-empty(uav) U altitude(uav) $\geq$ 7.0. □

When the operator is clear from context, we will often use the shorthand notation EXEC to refer to its associated execution flag fluent with default parameters. Using this notation, the formula above is written as EXEC U altitude(uav) $\geq$ 7.0.

Example 7.6.4 Whenever a UAV attaches to a box, it should become attached within 5000 ms and remain attached until explicitly detached. Using execution flags in an operator-specific monitor formula for the attach-box action, this can be expressed as follows:

$$\text{EXEC } \mathsf{U}_{[0,5000]}(\text{attached}(uav, box) \mathsf{U} \text{ executing-detach-box}(uav, box))$$

Compared to the motivational example, this formula is more efficient, since it is only tested when an attach action has been performed and only for the relevant UAV and box. The formula is also more robust, since failures will be signaled even if the box is dropped before flight altitude and regardless of the flight altitude of the UAV. □

7.7 Automatic Generation of Monitor Formulas

The use of a single logical formalism for modeling both planning and execution monitoring provides ample opportunities for the automatic generation of conditions to be monitored. Not only do actions have preconditions that must hold and effects that must take place, but it is also possible to analyze a complete plan and generate a set of links between actions, links where the effects of one action must persist over time until used as the precondition of one or more later actions. The ability to extract these conditions from a planning domain and transfer them to an execution monitor operating within the same formalism eliminates many potential sources of inconsistencies and inaccuracies.

Preconditions. Any operator is associated with a precondition formula ϕ. Given this formula, the operator-specific monitor condition ϕ can be generated: The precondition must hold immediately when the operator is invoked.

Prevail conditions. An operator can also be associated with a "prevail condition" formula stating a condition ϕ that must hold throughout the execution of

the action, as opposed to only holding in the first state. Then, the operator-specific condition $(\text{EXEC} \wedge \phi)\,\mathsf{U}\,\neg\text{EXEC}$ can be used.

Effects. The condition that the effect ϕ is achieved at some time during the execution of an action can be expressed using the monitor formula $\text{EXEC}\,\mathsf{U}\,\phi$. This, however, does not ensure that the effect still holds at the *end* of the action: It can be achieved at an intermediate point and then destroyed. The more elaborate formula $\text{EXEC}\,\mathsf{U}(\neg\text{EXEC} \wedge \phi)$ can be used to ensure that ϕ holds after the transition from execution to not executing. The formula $\text{EXEC}\,\mathsf{U}(\phi\,\mathsf{U}\,\neg\text{EXEC})$ can be used to express the same condition: The operator must execute until ϕ holds, after which ϕ must hold until the operator is no longer executing.

Temporal Constraints. An operator can be associated with constraints on the duration of its execution. Such constraints can be viewed in two different ways, both of which are supported by TALplanner: As a specification of the *most likely* behavior of the operator, which can be used to provide a reasonable estimate of the time required to execute a plan, or as a *definite* constraint on how much time can possibly be required if the operator is working as intended. In the latter case, the constraint can be used to generate a monitor formula constraining the amount of time that can pass before $\neg\text{EXEC}$ holds.

Causal Links. An effect monitor can ensure that the desired effects of an action materializes in the real world after the execution of the action. If this effect is later used as a precondition of another action, a precondition monitor can be used to ensure that the effect still holds – but between the effect and the precondition, a considerable amount of time may have passed. For example, **heli1** may use attach-box to pick up **bx3**, which gives rise to an effect monitor ensuring that attached(**heli1**, **bx3**) holds. Then, it may ascend to flight altitude, fly for several minutes towards its destination, descend, and use detach-box to put the box down. Only when detach-box is executed does the associated precondition monitor check that attached(**heli1**, **bx3**) is still true. If the UAV dropped the box during flight, it should have been possible to detect this much earlier.

This can be done by analyzing the complete plan and generating a set of causal links between different actions. In this example, such an analysis would have detected the fact that attached(**heli1**, **bx3**) is made true by attach-box, is required by detach-box, and is not altered by any action in between. Using execution flags, this global analysis can then attach the formula executing-attach-box(**heli1**, **bx3**) U (attached(**heli1**, **bx3**) U executing-detach-box(**heli1**, **bx3**)) to this specific instance of the attach-box operator in the plan.

It should be noted that this is highly dependent on having a sufficiently detailed description of the *intermediate* effects of any action: If an operator might change a value during its execution and then restores it before its end, the domain description must correctly model this in the operator description. In the example above, this would correspond to an operator which may temporarily put down **bx3** but is guaranteed to pick it up again. This should not be seen as a severe restriction: A rich and detailed domain model is also required for many other purposes, including but not limited to concurrent plan generation.

Alternatively, if not all intermediate effects are modeled but one has an upper

bound τ on how long an intermediate action may legitimately "destroy" a condition ϕ, a formula such as $\text{EXEC}1\ \mathsf{U}(\phi \wedge ((\neg\text{EXEC}2 \wedge \Diamond_{[0,\tau]}\,\phi)\ \mathsf{U}(\text{EXEC}2 \wedge \phi)))$ can be used: Operator 1 executes until it reaches a state where (a) the desired condition holds, meaning that the effect did take place, and (b) there is an interval of time where the operator 2 is not executing and where ϕ is always restored within τ units of time, followed by a state where operator 2 does execute and ϕ does hold.

7.7.1 Pragmatic Generation of Monitor Formulas

When one works on the automatic generation of monitor conditions, there is a strong temptation to generate every condition one can possibly find, without exception. After all, it would seem that the stronger the conditions one can place on execution, the better, and exceptions make algorithms more complex and less elegant. However in a system intended to operate in the real world, pragmatism is paramount, and one must take into account several reasons why some conditions should *not* be monitored: Some violations are not fatal, some information about the environment may be expensive or difficult to sense, and sensing may require special actions that interfere with normal mission operations. Additionally, the introduction of a richer and more detailed domain model should not automatically lead to heavier use of sensors.

For these reasons, our system is mainly built on the *selective* generation of conditions to be monitored: Each precondition, prevail condition, effect, and temporal constraint can be annotated with a flag stating whether it should be monitored. This provides most of the benefits of automatic formula generation while keeping the control in the hands of the domain designer.

7.8 State Generation

Monitor formulas are defined on TAL interpretations were each feature is associated with a value for every time-point. One way of viewing a TAL interpretation is as a sequence of states, where each state assigns a value to each feature. With this view, monitor formulas are defined on sequences of states, where each state provides a value for every feature that the formula refers to. Each state is also associated with a time-point, at which the values in the state are assumed to hold in the external world. In DyKnow terms, all values contained in a state should have the same valid time, which is also used as the valid time of the state itself.

DyKnow already has most of the necessary facilities for gathering the required information from distributed physical and virtual sensors, but so far, this information has been presented as a set of separate fluent streams with no synchronization between different streams. In this section, DyKnow will be extended with a state generation functionality that synchronizes a set of streams and generates a sequence of states satisfying the given constraints on valid times.

Definition 7.8.1 (State) A *state* in a knowledge processing domain D is a tuple of values in V_D. □

Definition 7.8.2 (State sample) A *state sample* in a knowledge processing domain D is a sample where the value is a state. □

Note that by this definition, any state is a value according to Definition 4.3.3 on page 37. We still prefer to introduce state as a new term, as there is a significant conceptual difference. A value is associated with a single feature and generally provides local information about a small part of a system. A state, on the other hand, is associated with multiple features. While it may be contextually generated and does not necessarily have to provide information about the entire system, it still tends to have a larger scope than a value.

Definition 7.8.3 (State stream) A *state stream* in a knowledge processing domain D is a stream where each stream element is a state sample. □

Ideally, all input streams used by the state generation function should generate values with identical valid times. If one input stream contains a value with valid time t, then every input stream contains a value with valid time t. If this holds, the state generator merely has to wait for the desired readings to propagate through the distributed system, caching feature values as they arrive. As soon as all fluent streams have generated values with valid time t, a complete state with valid time t can be produced.

In practice, one cannot expect all physical and virtual sensors to be fully synchronized, especially if they belong to separate platforms in a distributed system without central control. Also, communication failures may lead to missing values in some fluent streams. Any state generation mechanism must be able to take this into account and generate states for time-points where only partial information is provided by input streams.

One can define a variety of different policies determining how states are synchronized and reconstructed from partial information. For example, one could decide to create a new state for each time-point t where any input fluent stream has a value and assume that the most recent value is still valid for those fluent streams without a sample with valid time t. Another example would be to wait until new values have arrived in all input streams before creating a new state. Again, we would have to assume that the most recent value is still valid for those fluent streams that do not have a sample with valid time t. Which approach to generating states is most suitable depends on the application. We therefore introduce a *state synchronization policy* which describes the desired properties of a state stream.

Example 7.8.1 (State stream example) Let us continue Example 7.5.1 on page 109 where we would like to monitor the condition that whenever a UAV is moving, the winch should not be lowered. To evaluate such a formula we need to create a stream of states containing the two features speed(*uav*) and winch(*uav*) for each *uav*. In this example we have a single UAV called uav1. An example state for these features is $\langle 29.7, 31.8 \rangle$, where 29.7 is the value of speed(uav1) and 31.8 is the value of winch(uav1).

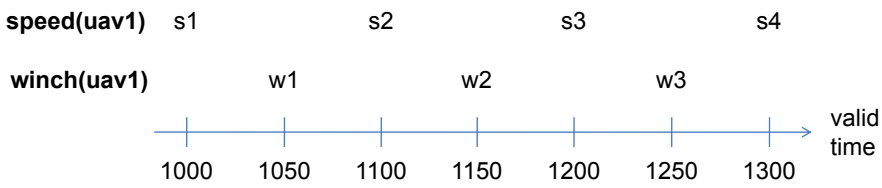


Figure 7.8: An example of two fluent streams with the same sample period, 100 time units, which are not synchronized. The values are ordered by their valid time.

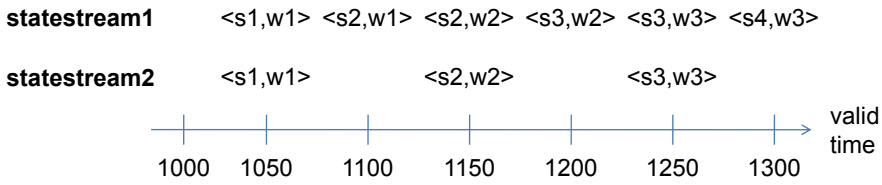


Figure 7.9: Two possible state sequences that can be generated from the fluent streams in Figure 7.8.

In our architecture this is an example where the necessary information comes from two sensors which are not synchronized. The speed of the UAV is estimated by the helicopter state estimation functionality on the PFC, while the state of the winch is provided by another sensor. Even if the two sensors are sampled with the same frequency, it is not guaranteed that they will be sampled at the exact same time-points. Therefore we might, for example, get the two fluent streams shown in Figure 7.8. Both fluent streams have a sample period of 100 time units, but they are not synchronized since they are sampled at different time-points.

From the fluent streams in Figure 7.8 several possible state sequences can be generated. Two of these are shown in Figure 7.9. The first state stream is generated using the first example policy, where a state is created each time at least one of the input streams has changed values, while the other is generated using the second example policy by creating a state when all the input streams have changed values. Other state streams could be conceived of as well. This shows that it is not obvious how to extract states from two or more fluent streams. Each state stream must therefore be generated according to a state synchronization policy which states which of the possible state sequences should be generated. □

7.8.1 A Basic State Generation Algorithm

In this section we describe a basic algorithm for generating a state sequence in the form of a state stream by synchronizing a set of fluent streams. Fluent stream synchronization can be seen as a function on streams, as shown in Figure 7.10, which is very much like a computational unit. The synchronization function is specified by a state synchronization policy, which provides a standardized way for any

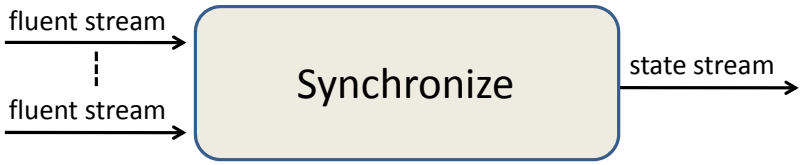


Figure 7.10: A conceptual view of synchronizing a set of fluent streams in DyKnow.

knowledge process to generate state sequences satisfying a wide variety of different criteria. If a particular application requires a synchronization function which is not supported by the provided state synchronization policies, then a computational unit can be used as a general mechanism for extracting state streams from fluent streams.

There are two main types of algorithms for synchronizing streams. The first type is based on sampling where the input streams are synchronized at periodic time-points, like a sampled fluent stream. The second type works asynchronously where a state is generated each time the input streams have changed sufficiently to count as a new state. The definition of what counts as a sufficient change would be part of the state synchronization policy. In this thesis we present algorithms for state synchronization based on sampling according to state synchronization policy specifications, defined as follows.

Definition 7.8.4 (State synchronization policy specification) A *state synchronization policy specification* for a KPL signature σ has the form du, sa, va, dba, de, where du is a duration constraint specification for σ, sa is a sample change constraint specification for σ, va is an approximation constraint specification for σ, dba is a delay before approximation constraint specification for σ, and de is a delay constraint specification for σ. □

The duration, change, approximation, and delay constraint specifications are defined in Section 4.4.5 on page 52. The delay before approximation constraint is defined as follows.

Definition 7.8.5 (Delay before approximation constraint) A *delay before approximation constraint* for a KPL signature $\sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{N}, \mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{V} \rangle$ has the form **delay before approximation** d, where d is a time-point symbol in $\mathcal{T}$. □

Basic State Synchronization

A straight-forward way of creating a sampled state stream is to query each fluent stream for its most recent value at each of the sampling time-points. For example, we could sample the fluent streams in Figure 7.8 every 100 time units starting at time-point 1050. However, Figure 7.8 only shows the valid time for each sample. Due to communication and processing delays, these samples are not instan-

speed(uav1)			winch(uav1)		
value	valid time	available time	value	valid time	available time
s1	1000	1010	w1	1050	1120
s2	1100	1110	w2	1150	1170
s3	1200	1280	w3	1250	1260
s4	1300	1320			

Table 7.1: The valid and available times for each sample in the example speed(uav1) and winch(uav1) fluent streams.

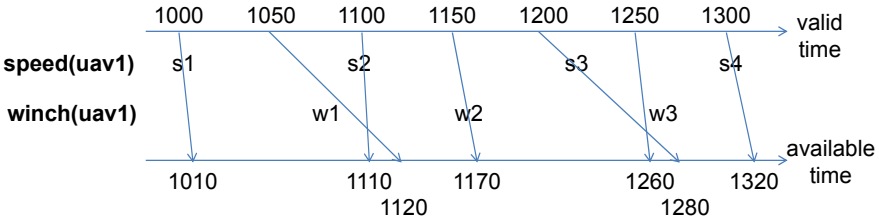


Figure 7.11: The same fluent streams as in Figure 7.8 on page 119 but with both valid and available times shown.

taneously available to the synchronization algorithm. It is therefore necessary to take into account the available time of each sample, which is the actual time when the sample becomes available in the fluent stream.

Table 7.1 extends the example with available times and Figure 7.11 visualizes the same fluent streams with a dual time line. For example, the value w1 is valid at time 1050 but only arrives at time 1120. Two crossing arrows in the figure means that a sample arrives later than the sample whose arrow it crosses even though it has an earlier valid time.

We now see that if the fluent streams in Figure 7.11 are queried for the latest available value every 100 time units starting at time-point 1050, we would get a state stream containing the states $\langle$s1, no_value$\rangle$, $\langle$s2, w1$\rangle$, $\langle$s2, w2$\rangle$, and $\langle$s4, w3$\rangle$. Due to the delays we did not get the expected statestream2 as shown in Figure 7.9.

If a maximum delay d is known in advance, the state generation function can wait until time $t + d$ before sampling the input streams and generating a state with valid time t. This guarantees that all samples have had enough time to arrive, but still does not give the intended result. Suppose, for example, that the maximum delay for the streams in Figure 7.11 is 80 time units, which also happens to be the actual delay for the value s3. In order to generate states with valid times every 100 time units starting at $t = 1050$, one would then sample every 100 time units starting at $t + d = 1130$. The result would be a state stream containing the states $\langle$s2, w1$\rangle$, $\langle$s2, w2$\rangle$, and $\langle$s4, w3$\rangle$. The reason we still do not get the expected result is that we query each fluent stream for its *most recent* value at $t+d$. If a value is not delayed with the maximum delay then a value with a later valid time than t might be available in the fluent stream at $t + d$. For example, the value s2 with the valid

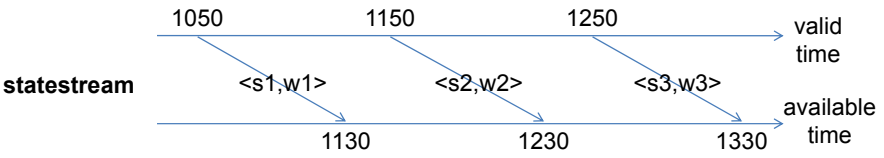


Figure 7.12: The state sequence generated from the fluent streams in Figure 7.11 on the preceding page by the basic state synchronization algorithm.

time 1100 is available at 1110 with only a delay of 10 time units. To handle this case we need to query a fluent stream for the sample with the greatest *valid* time less than or equal to t, among those samples which are available at $t + d$.

To get the most recent value in a fluent stream f for a particular time-point t among the samples that are available to the fluent stream at time-point t_q we use the function $most\_recent\_at(f, t, t_q)$ from Definition 4.3.11 on page 42. The function is defined to return the sample with the highest valid time less than or equal to t among those samples with an available time less than or equal to t_q. The time-point t_q is called the *query time*. For example, if we query the speed(uav1) fluent stream at time-point 1130 asking for the last sample whose valid time is at or before 1050, $most\_recent\_at$(speed(uav1), 1050, 1130), we get s1.

Using $most\_recent\_at$ it is possible to query each fluent stream for the most recent value at each sampling time-point t with the query time $t + d$. The state at time-point t for the fluent streams $f_1, \ldots, f_n$ is $\langle most\_recent\_at(f_1, t, t + d), \ldots, most\_recent\_at(f_n, t, t + d)\rangle$. This results in the state stream shown in Figure 7.12, which is the same as statestream2 in Figure 7.9 on page 119.

It is not enough to know the maximum delay to make the basic algorithm work, we must also know at what time-points to sample the input fluent streams. To determine these time-points we will use the policies of the input streams. Assume we would like to create a state when at least one of the input streams has changed values using the most recent value approximation strategy.

The first observation is that no state can be computed until each input fluent stream has at least one sample. This means that the earliest valid time for the state stream is the latest first valid time of any of the input streams. As long as the input streams are not removed, new states can be generated until no further samples are generated by them. This means that the latest valid time for a state stream is the latest valid time of any of the input streams. The duration constraint of the policy of a fluent stream specifies both the first valid time, the start time, and the last valid time, the end time. It is therefore possible to derive the maximal duration constraint of the state synchronization policy from the duration constraints of the input streams.

What is more interesting is at what intermediate time-points states should be computed. To catch all possible state changes we have to compute a state for each valid time where at least one input fluent stream has changed its value. If all input streams are sampled, then this is determined by their sample periods as specified by their policies. To get the desired sample period of the state stream, we compute

the greatest common divisor of all the sample periods of the input fluent streams. If they are not sampled, but we still want to use a sampled state synchronization policy, then we either have to sample at each time-point or use an ad-hoc sample period which is considered sufficient.

Example 7.8.2 (State stream example (continued)) To synchronize the fluent streams in Example 7.8.1 on page 118, the state synchronization policy should contain the constraints "**sample every** 100, **from** 1050 **to** 1250, **max delay** 80". The resulting state sequence using this synchronization policy is shown in Figure 7.12.

If the sample period of one of the input fluent streams would have been 75 instead of 100, then the sample period of the resulting state stream would be 25 since this is the greatest common divisor of 75 and 100. □

If the maximum delay and the sample period of each input fluent stream is known then this approach will generate a state stream which contains all the changes in any of the input streams. If the maximum delay is not known then a suitable value has to be chosen by the user in order to make a trade-off between accuracy and timeliness. However, there is more information available in the fluent stream constraints that could be used in order to improve the algorithm by reducing the delays and handling asynchronous streams.

7.8.2 An Improved State Generation Algorithm

To improve the basic state synchronization approach we have to compute the same states as it does, but earlier. To compute a state at a particular valid time the algorithm needs all the information from the input streams which is relevant for this time-point. The issue is therefore to determine when all the available information has been received.

In the general case, samples can arrive in an arbitrary order. Therefore, if a sample with valid time t arrives it does not say anything about whether more samples with earlier valid times are going to arrive or not. However, if the input fluent stream has a monotone order constraint and a sample with valid time t arrives, then we know that all samples with valid times earlier than t have arrived. With a strict monotone order constraint we know that all samples with valid times up to and including t for this input stream have arrived. It should be noted that if a stream is sampled then it implicitly has a strict monotone order constraint. A state synchronization algorithm only requiring the monotone order constraint for its input streams is therefore more general than one requiring a sample constraint.

If all input streams have the strict monotone order property and we have received samples with valid times at or after t from each of them, then we could compute the states up to t because we know we have all the information that will be available. For example, given the fluent streams in Figure 7.11 on page 121 the three states with valid times 1050, 1150, and 1250 can be computed at time-points 1120, 1170, and 1280, as compared to 1130, 1230, and 1330 using the basic approach. The higher the maximum delay is compared to the average delay, the more potential there is to reduce the delay of the synchronized states.

The improved algorithm used by DyKnow to synchronize states is described in two steps. First we describe how to synchronize two or more fluent streams at a specific time-point and then how to generate these synchronization time-points.

Synchronize at a Time-Point

This section describes an algorithm for synchronizing a set of fluent streams at a particular time-point called the *synchronization time*. This time-point is not determined by the algorithm but must be given, for example by a sample constraint as explained later.

The idea behind this algorithm is to classify each input stream according to the information which is available about the synchronization time. To classify an input stream three properties are used. The first property is whether the input stream contains, among the samples received so far, a sample at the synchronization time, i.e. a sample with valid time equal to the synchronization time. The second property is whether an approximation is available given the samples received so far. This property depends on the value approximation constraint associated with each input stream. In the case of a most recent value approximation constraint the input stream must contain a sample with a valid time before the synchronization time in order to have an approximation available. If instead a linear approximation constraint is used then at least two samples, one with a valid time before and one with a valid time after the synchronization time, are needed. The third property is whether more information about the synchronization time might become available. For example, if the maximum delay is 100 then at time $t + 100$ all the information about t is known, but up to that time more information might become available.

Using these three properties each input stream is classified as being in one of five categories at every time-point:

- EXACT, exact value available,
- APRXFINAL, approximation available and no further information expected,
- APRXMORE, approximation available and further information may arrive,
- NOAPRXFINAL, no approximation available and no further information expected, or
- NOAPRXMORE, no approximation available and further information may arrive.

Using these categories it is possible to determine if a state can be computed at the synchronization time or not. We will consider four different cases depending on which set of categories cover the set of input streams: All input streams are in categories EXACT or APRXFINAL, some input stream is in category APRXMORE while the rest of them are in categories EXACT or APRXFINAL, no input stream is in category NOAPRXFINAL while some input stream is in category NOAPRXMORE, and some input stream is in category NOAPRXFINAL.

In the first case, where all input streams are in categories EXACT or APRXFINAL it is possible to compute a state at the synchronization time. Further, if all the input streams are in category EXACT then we have a state without any approximation. Since no more information will become available about the synchronization time-point this is the best possible state.

In the second case, where some input stream is in category APRXMORE while the rest of them are in categories EXACT or APRXFINAL it is possible to compute an approximated state at the synchronization time-point. However, since at least one of these input streams is in category APRXMORE an exact value or a better approximation might become available later. To handle the trade-off between using an existing approximation and waiting for a better one, a *delay before approximation* parameter is introduced. The parameter determines how long after the synchronization time the algorithm will wait for a better approximation. This means that the algorithm will wait either until time-point t + *delay before approximation* or until all input streams are in categories EXACT and APRXFINAL.

Example 7.8.3 (State stream example (continued)) Consider synchronization at time-point 1150 of the speed(uav1) and winch(uav1) fluent streams using a most recent value approximation constraint. When s2 with valid time 1100 is available in the speed(uav1) fluent stream at time-point 1110 the input stream will be in categoryAPRXFINAL, since according to the sample constraint we know that no new value will become available with a valid time of less than 1150 (the next value will have a valid time of 1200). When the first sample in the winch(uav1) fluent stream is available at time-point 1120 with the value w1, we can approximate the value at 1150 to w1 using the most recent value approximation constraint. However, since we know by the sample constraint that the next sample should have a valid time of 1150 further information is expected and the second input stream is therefore in category APRXMORE. If the *delay before approximation* parameter is set to a very low value of 10 the algorithm will only wait until time-point 1160 when the value of winch(uav1) would be approximated to w1. If the parameter is set to 20 or more, then the next winch(uav1) sample will have time to arrive at 1170, both input streams will be in category EXACT or APRXFINAL, and the best possible state will be computed. □

In the third case, no input stream is in category NOAPRXFINAL while some input stream is in category NOAPRXMORE which means that it is not yet possible to approximate a state at the synchronization time-point. However, since some input stream is category NOAPRXMORE more information might become available later. Since there is no guarantee that more information will become available it is necessary to introduce a maximum waiting time before accepting that no state can be computed for this synchronization time-point. This is handled in a similar way as input streams in category APRXMORE, by using a parameter called *maximum delay*. This means that the algorithm will wait at most *maximum delay* time units for an approximation to become available for a synchronization time-point, after which no state can be approximated given the current parameters.

In the fourth and final case, some input stream is in category NOAPRXFINAL which means that it is not possible to approximate a state. Further, since no new information is expected no state can ever be approximated at the time-point with the current parameters.

The problematic cases are when either some input streams are in category NOAPRXFINAL or some input streams are in category NOAPRXMORE and the maximum delay has been waited. In these cases it is not possible to approximate any state given the current parameters. In this situation there are basically two choices, either give up and notify the system about the failure to produce a state at the synchronization time-point or make some kind of approximation of the state anyway using the available information.

If the state synchronization policy contains a sample change constraint, then failing to compute a state would result in a violation of the policy. Therefore we choose to do the best given the current information. In this case we will use the constant `no_value` as the value of those input streams where no proper approximation can be made. This means that a state will always be computed for every synchronization time-point, but it may contain `no_value` constants.

The reason for a failure to compute a state could for example be due to too small accepted delays. Since the algorithm will try to minimize the delay it is quite safe to set a high maximum delay. The only case when this is problematic is when no information will ever arrive for a particular time-point, since then it will take a long time before this is detected.

To inform the algorithm when it is time to consider inputs in categories APRXMORE and NOAPRXMORE two timers are created. The first timer is set to go off *delay before approximation* time units after the synchronization time-point and the other *maximum delay* time units after the synchronization time-point. This means that there are two different events which the state synchronization algorithm has to react to. The first is when a new sample arrives from one of the input fluent streams. The second is when one of the timers generates a timeout. The pseudo code for the top level of the algorithm is as follows:

```
1  procedure sync_at_time_point(t, delay_before_approx, max_delay)
2  sync_time ← t
3  set_timeout_at(sync_time + delay_before_approx)
4  set_timeout_at(sync_time + max_delay)
5   do
6    wait for input or timeout
7    if received sample s from input stream i at time t  then
8      update_after_input(t, i, s)
9    elsif received timeout t  then
10     update_after_timeout(t)
11 while not synchronized(t)
12 compute state at sync_time
```

Since the synchronization algorithm does not use each sample directly when it arrives there is a need to buffer samples. Each input fluent stream is associated with its own buffer. When a new sample is received from one of the input streams the

corresponding buffer is updated. The buffer is updated by adding the new sample, throwing away obsolete samples that are no longer useful, and computing its new category. A sample is no longer useful when it can not be used either as an exact value or to approximated a value for the input stream. When this happens depends on how values are approximated. In the case of a most recent value approximation, then only a single value before the synchronization time-point has to be stored and all previous samples can be discarded.

```
1 procedure update_after_input(t, i, s)
2 add sample s to buffer i
3 remove obsolete samples from buffer i
4 update the category for buffer i
```

To determine if more information might be available for a specific time-point the buffer uses the policy of the input stream to determine if there is a maximum delay or if the samples are ordered by valid time. If the maximum delay is d and the time now is t then all information up to $t - d$ have arrived. If the input fluent stream has a strict or non-strict monotone order constraint or a sample constraint then samples are ordered by valid time, and all information up to the valid time of the last sample has been received. If the input stream neither has a maximum delay nor is ordered by valid time then the buffer will never be able to conclude that no more information about a time-point will become available. In this case the algorithm will always wait *delay before approximation* time units before approximating a value.

When a timeout arrives it means that either more than *delay before approximation* or *maximum delay* time units have passed since the synchronization time. Since the passing of time might affect the category of each of the buffers they have to be recalculated before checking if a state can be computed or not.

```
1 procedure update_after_timeout(t)
2 for each buffer i do
3   update the category for buffer i
```

To check whether it is possible to create a synchronized state at the synchronization time-point given the samples currently in the buffers we need to check which categories the buffers are in. The following procedure does that and returns `true` if a state can be computed at the synchronization time otherwise it will return `false`:

```
1 procedure synchronized(now)
2 for each input stream i in category NOAPRXFINAL do
3   approximate i with no_value
4   set the category of i to APRXFINAL
5 if all input streams are in categories EXACT or APRXFINAL then
6   return true
7 elsif all input streams are in categories EXACT, APRXFINAL, or APRX-
  MORE and sync_time + delay_before_approx ≤ now then
8   return true
9 elsif sync_time + max_delay ≤ now then
```

```
10    for each input stream i in category NoAprxMore do
11      approximate i with no_value
12      set the category of i to AprxFinal
13    return true
14  else
15    return false
```

The procedure starts by handling those input streams which are in category NoAprxFinal, which means that they have no approximation at the synchronization time-point and no more information related to that time-point is expected. Since no proper approximation is possible the input stream is approximated with the constant `no_value`. The procedure then checks the three cases when the input streams are synchronized at the synchronization time-point and a state can be computed. In the first case, we either have an exact observation or an approximation with no further information available for each input at the synchronization time. In the second case, the delay before approximation has passed and a state is computed if every input stream has either an exact observation or an approximation at the synchronization time-point, even though more information might arrive later. In the third case, the maximum delay has passed and each of the input streams in category NoAprxMore will be approximated with the constant `no_value`. If none of the cases apply then no synchronization at the synchronization time-point has been achieved and the procedure returns `false`.

Sampled Synchronized State Stream

In order to create a sampled state stream the sync_at_time_point procedure has to be called with successive time-points, one for each of the time-points to sample. To generate these time-points, start at the start time and after a state has been created or rejected update the synchronization time-point by increasing it with the sample period. To specify a stream generator of such state streams, a *state stream generator declaration* is used.

Definition 7.8.6 (State stream generator declaration) A *state stream generator declaration* for a KPL signature σ has the form **strmgen** l = **state**$(f_1, \ldots, f_n)$ **with** p, where l is a label term for σ, $f_1, \ldots, f_n$ are fluent stream terms for σ, and p is a state synchronization policy specification for σ. □

To generate the input to the state stream generator labeled s according to the state stream generator declaration "**strmgen** s = **state**$(f_1, \ldots, f_n)$ **with from** *start_time* **to** *end_time*, **sample every** *sample_period*, **use most recent**, **delay before approximation** *delay_before_approx*, **max delay** *max_delay*" the following algorithm is used:

```
1  procedure sampled_synchronized_state_stream( f_1, ..., f_n, start_time,
     end_time, sample_period, delay_before_approx, max_delay)
2  for each f_i do
3    create a buffer and set up a subscription to f_i
```

4 *sync_time* ← *start_time*
5 **while** *sync_time* ≤ *end_time* **do**
6 sync_at_time_point(*sync_time*, *delay_before_approx*, *max_delay*)
7 *sync_time* ← *sync_time* + *sample_period*

A State Generation Example

This section continues Example 7.8.1 on page 118 and shows how the synchronization algorithm extracts a state sequence from the speed(uav1) and winch(uav1) fluent streams as shown in Figure 7.11 on page 121. Both input fluent streams have the constraints "**sample every** 100" and "**max delay** 100". The state fluent stream that will be generated has the constraints "**from** 1050 **to** 1250, **sample every** 100, **delay before approximation** 100, **max delay** 100". To approximate values the most recent value approximation constraint is used, which means that to approximate the value at time-point *t* the sample with the latest valid time before *t* is used.

The internal state and the states generated in each step of the state generation algorithm as the content of the fluent streams become available are shown in Table 7.2 on the next page. The first column is the current time when a sample becomes available. The second column is the current synchronization time-point. The next four columns show the buffer and the category for the two input streams. The seventh and final column shows the state generated by the algorithm at the current step, if any.

The first row in Table 7.2 on the following page shows the initial state of the synchronization algorithm. The current time is 1000, the initial synchronization time-point is 1050 as stated by the duration constraint in the state synchronization policy, and both fluent stream buffers are empty. Since the buffers are empty but more information is expected they are both in category NOAPRXMORE.

In the second row the internal state of the algorithm at time 1010 is shown, when the first sample for speed(uav1) with value s1 and valid time 1000 is available. Since there is a value before the synchronization time of 1050 an approximated value exists, and since the sample period is 100 the next value is expected to have a valid time of 1100 which means that no more information related to time-point 1050 is expected. Therefore speed(uav1) is now in category APRXFINAL. Since no approximation is available for the second fluent stream no state can be extracted at time-point 1010.

As can been seen in Figure 7.11 on page 121 the next sample from the speed(uav1) fluent stream will also arrive before the first sample from winch(uav1) is available. The third row shows the internal state of the algorithm after this second sample from speed(uav1) has become available. The situation is almost the same as before. The only difference is that the buffer now contains two samples.

The fourth row shows the internal state of the synchronization algorithm when the first sample for winch(uav1) has arrived at 1120. Since the valid time of the sample is the same as the synchronization time the buffer is placed in category EXACT and since both input streams now are in either category EXACT or APRXFINAL the state ⟨s1, w1⟩ can be extracted.

sync	speed(uav1)		winch(uav1)		
now	time	buffer	category	buffer	category	new state
1000	1050	{}	NoAprxMore	{}	NoAprxMore	-
1010	1050	{⟨1000, 1010, s1⟩}	AprxFinal	{}	NoAprxMore	-
1110	1050	{⟨1000, 1010, s1⟩, ⟨1100, 1110, s2⟩}	AprxFinal	{}	NoAprxMore	-
1120	1050	{⟨1000, 1010, s1⟩, ⟨1100, 1110, s2⟩}	AprxFinal	{⟨1050, 1120, w1⟩}	Exact	⟨1050, 1120, ⟨s1, w1⟩⟩
1120	1150	{⟨1100, 1110, s2⟩}	AprxFinal	{⟨1050, 1120, w1⟩}	AprxMore	-
1170	1150	{⟨1100, 1110, s2⟩}	AprxFinal	{⟨1150, 1170, w2⟩}	Exact	⟨1150, 1170, ⟨s2, w2⟩⟩
1170	1250	{⟨1100, 1110, s2⟩}	AprxMore	{⟨1050, 1120, w1⟩}	AprxMore	-
1260	1250	{⟨1100, 1110, s2⟩}	AprxMore	{⟨1250, 1260, w3⟩}	Exact	-
1280	1250	{⟨1200, 1280, s2⟩}	AprxFinal	{⟨1250, 1260, w3⟩}	Exact	⟨1250, 1280, ⟨s3, w3⟩⟩

Table 7.2: The internal state of the synchronization algorithm and its output during the generation of a state stream from the speed(uav1) and winch(uav1) fluent streams as shown in Figure 7.11 on page 121.

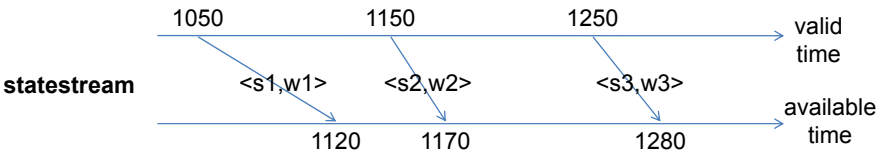


Figure 7.13: The state stream generated by the improved state synchronization algorithm from the fluent streams in Figure 7.11 on page 121.

Since a state has been extracted for the current synchronization time-point, the synchronization time is updated by adding the sample period of 100. The internal state of the synchronization algorithm with the new synchronization time-point is shown as the fifth row. The first sample in the speed(uav1) buffer is removed since it is no longer relevant due to the choice of approximation strategy and the fact that a sample with a valid time closer to the new synchronization time is available.

The sixth and seventh rows show the state of the synchronization algorithm as the second state is extracted at time-point 1170 when the second sample becomes available in winch(uav1).

Finally, rows eight and nine show how the third and final state is extracted when the last two samples arrive, one for each of the input streams. This is the final state extracted since the end time of the state stream is 1250.

The state sequence generated using the improved algorithm is shown in Figure 7.13. It is the optimal state sequence that could be generated from these fluent streams, i.e. the state sequence which captures all changes where any input fluent stream has changed values and does it with the minimal delay.

In this example, the algorithm did not have to use any approximated values because the delay before approximation timer had expired. In fact this can never happen if the delay before approximation is equal to or greater than the maximum delay of the input streams, which it was in this case. However, if we change the delay before approximation to 70 then the third state extracted would have been $\langle s_2, w_3 \rangle$, since the third speed(uav1) sample is delayed 80 time units.

7.9 Execution Monitoring with Inaccurate Sensors

The purpose of execution monitoring is the detection of such failures that would prevent a system from achieving its designated goals, or that would cause other types of undesirable or potentially dangerous behavior. In this process, one should not only work to maximize the probability that a failure is detected but also attempt to minimize the probability of false positives. In other words, a system should not signal a failure if none has occurred. In some cases this may be even more important than detecting non-catastrophic failures, because while such failures can prevent a system from achieving all its subgoals, a persistent false positive could cause the system to stall entirely, believing it is continuously failing.

There are several different reasons for false positives, and different approaches may be suitable for dealing with these problems. There is of course always the

possibility of *catastrophic sensor failure*, where a sensor begins returning nonsensical information. Though this can perhaps be modeled and detected, we consider it outside the scope of the thesis. Instead, we focus on the difficulties that are present even when sensors are functioning nominally. Our architecture being a distributed system, there is always a possibility of *dropouts* and *delays* where sensor data temporarily disappears due to a communication failure. Similarly, events may propagate through different paths and arrive *out of order*. Also, one must always expect a certain amount of *noise* in sensor values.

These difficulties can be ameliorated through careful state generation. Temporary dropouts can be handled through extrapolating historical values. Delays can be handled by waiting until time $n + m$ before progressing a formula through the state at time n; sensor values from time n then have an additional m milliseconds to propagate before the state generator assumes that all values must have arrived. This obviously has the cost of also delaying failure detection by m milliseconds, which requires a careful consideration of the compromise to be made between prompt detection and the probability of false positives. Noise could conceivably be minimized through sensor value smoothing techniques and sensor fusion techniques where the measurements from several sensors are taken into account to provide the best possible estimation.

Though additional measures could be considered, it follows from the fundamental nature of a distributed system with noisy sensors that the possibility of inaccuracies in the detected state sequence can never be completely eliminated. Instead, we suggest a two-fold approach to minimizing false positives: Be careful when generating states, but also be aware that state values may be inaccurate and take this into consideration when writing monitor formulas.

Consider, as an example, the condition $\Box \forall uav.\mathsf{speed}(uav) \leq T$. On the surface, the meaning of this formula would seem to be that the speed of a UAV must never exceed the threshold T. However, this formula will not be evaluated in the real world: It will be evaluated in the state sequence approximation generated by DyKnow, and there, its meaning will be that the *sensed and approximated* speed of a UAV must never exceed the threshold T. Since a single observation of $\mathsf{speed}(uav)$ above the threshold might be an error or a temporary artifact, a more robust solution would be to signal a failure if the sensed speed has been above the threshold during an interval $[0, \tau]$ instead of at a single time-point. This can be expressed as $\Box \Diamond_{[0,\tau]} \mathsf{speed}(uav) \leq T$: It should always be the case that within the interval $[0, \tau]$ from now, the sensed speed returns to being below the threshold.

Since the formula above only requires that a single measurement in every interval of length τ must be below the threshold, it might be considered too weak for some purposes. An alternative would be to require that within τ time units, there will be an *interval* of length τ' during which the UAV stays within the limits: $\Box\, \mathsf{speed}(uav) > T \rightarrow \Diamond_{[0,\tau]} \Box_{[0,\tau']} \mathsf{speed}(uav) \leq T$.

7.10 Empirical Evaluation of the Formula Progressor

For our approach to be useful, it must be possible to progress typical monitor formulas through the state sequences generated during a mission using the often limited computational power available in an autonomous robotic system, such as the DRC computer on board our mobile UAV platforms. Early testing in our emergency services planning domain, both in flight tests and in hardware-in-the-loop simulation, indicated that the progression system had more than sufficient performance for the formulas we used, requiring only a fraction of the available processing power. If there is a bottleneck, then it lies not in the use of formula progression but in actually retrieving and processing the necessary sensory data, which is necessary regardless of the specific approach being used for monitoring execution and is therefore outside the purview of this chapter.

Nevertheless, the processing requirements for the progression of typical monitor formulas should be quantified in some manner. Our evaluations build on experiments with the emergency services planning domain as well as the use of synthetic tests. For the remainder of this section, we will focus on the synthetic tests, where we can study highly complex combinations of time and modality. We use the actual DRC computer on board an RMAX UAV to run progression tests for formulas having forms that are typical for monitor formulas in our application. State sequences are constructed to exercise both the best and the worst cases for these formulas.

Results are reported both in terms of the average time required to progress a certain number of formulas through each state in a sequence and in terms of how progression time changes across states in the sequence, due to formula expansion during progression. This can be directly translated into the number of formulas that can be progressed on the DRC computer given a specific state sampling rate. The final objective, of course, is not to progress a vast number of formulas but to ensure that the required formulas can be progressed using only a small percentage of the CPU, leaving enough time for other processes and services to run.

Depending on the domain and the execution properties that should be modeled, the number of formulas that need to be progressed concurrently will vary. In our work with the emergency services application, we observed the use of from one or two up to at most a few dozen formulas at any given time in the planning context, with the average being towards the low end of the scale. Since the execution monitor is a system service, other services may also have monitoring requests, so many more formulas may need to be progressed concurrently.

In all experiments, states were generated with a sampling period of 100 ms.

7.10.1 Experiment: Always Eventually

In the first experiment, we used a formula of the common form $\Box \Diamond_{[0,1000]} p$, where p is a single predicate, corresponding to the fact that p must never remain false for more than one second. This was progressed through several different state sequences, carefully constructed to exercise various progression behaviors. In the

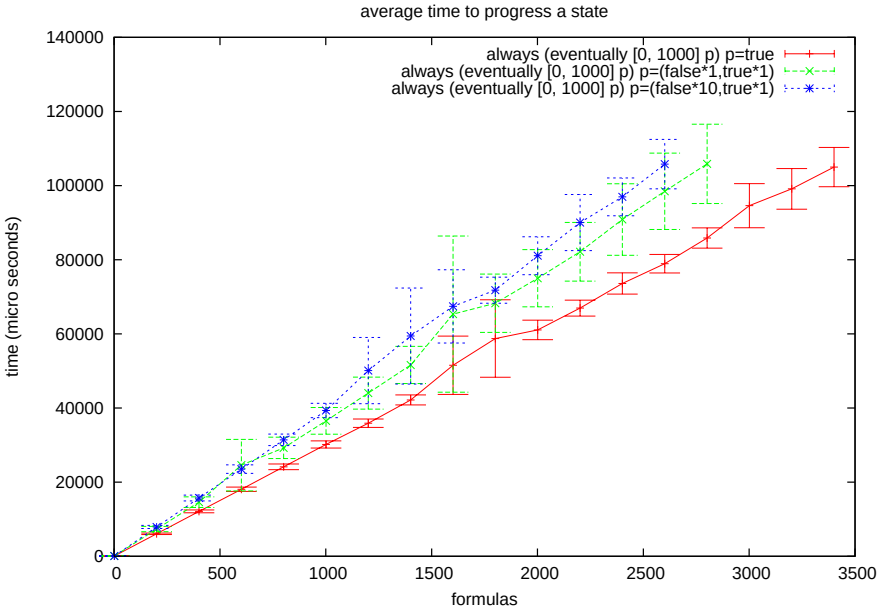


Figure 7.14: Testing: Always Eventually (average progression time).

following, let $\phi = \Box \Diamond_{[0,1000]} p$.

- **(true)** – p is always true. This is the best case in terms of performance, since each time $\Diamond_{[0,1000]} p$ is progressed through a state, it immediately "collapses" into $\top$. What remains to evaluate in the next state is the original formula ϕ.
- **(true,false)** – p alternates between being true and being false. This is an intermediate case, where every "false" state results in the formula $\Diamond_{[0,1000]} p$ being conjoined to the current formula, and where this subformula collapses into $\top$ in every "true" state.
- **(false*10,true*1)** – p remains false for 10 consecutive sampling periods (1000 ms), after which it is true in a single sample. The sequence then repeats. Had p remained false for 11 consecutive samples, the formula would have been progressed to $\bot$, a violation would have been signaled, and progression would have stopped. Consequently, this corresponds to the worst case.

The results shown in Figure 7.14 can be expressed in a number of ways. If the sample period is 100 ms, all formulas must be progressed within 100 ms or the progressor will "fall behind" as new states arrive. In this situation, approximately 2500 formulas can be used even with the worst case state sequence. From another perspective, if there will be a maximum of 10 formulas of this type to progress

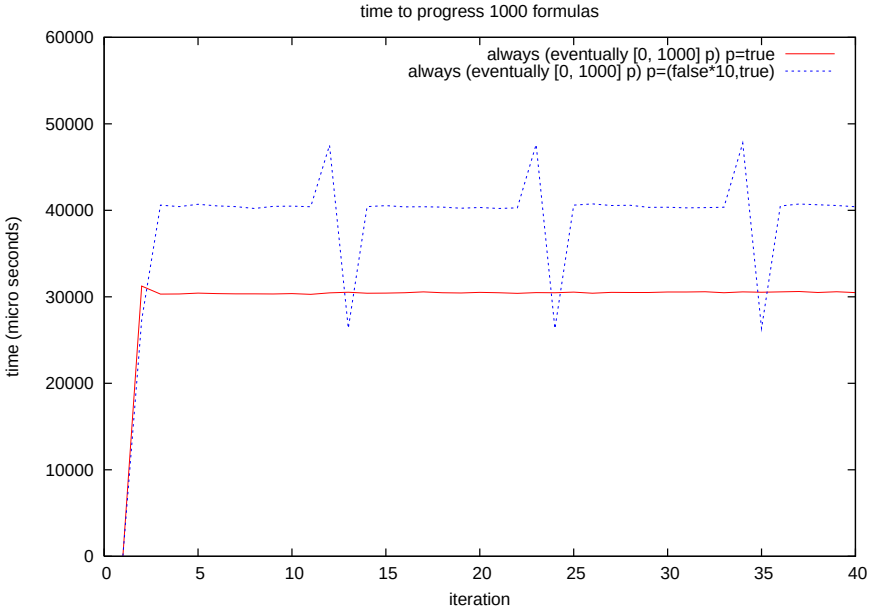


Figure 7.15: Testing: Always Eventually (development over time).

at any given time, then this will require up to 0.4% of the CPU time on the DRC computer.

Results will also vary depending on the complexity of the inner formula inside the tense operators. Though the figure refers to tests run using a single fluent p, even the most complex inner formulas used in the UAV logistics domain require only a small constant multiple of the time shown in this graph.

It should be noted that the time required to progress a formula through a state is not always constant over time. The formulas themselves may vary as they are progressed through different states, and time requirements for progression vary accordingly. The average progression time must be sufficiently low, or the progressor will fall behind permanently. The *maximum* progression time should also be sufficiently low, or the progressor will *temporarily* fall behind. Figure 7.15 shows the precise time requirements for progressing 1000 instances of the formula $\Box \Diamond_{[0,1000]} p$ through each numbered state sample, using two different state sequences.

- In the best case, where p is always true, progression time is constant at around 30 μs per formula for a state sequence where p is always true. This is exactly what would be expected: The result of progressing the formula $\Box \Diamond_{[0,1000]} p$ through a state where p is true should always be the formula itself.

- In the worst case, where p is false for 10 sample periods and then true,

$\Box \Diamond_{[0,1000]} p$ is initially progressed to the somewhat more complex formula $\Diamond_{[-100,900]} p \wedge \Box \Diamond_{[0,1000]} p$, indicating that p must become true within 900 ms. In the next state, p remains false, and the formula is further progressed to $\Diamond_{[-100,900]} p \wedge \Diamond_{[-200,800]} p \wedge \Box \Diamond_{[0,1000]} p$. This formula states that p must become true within 900 ms *and* within 800 ms, which can be reduced to the statement that it must become true within 800 ms: $\Diamond_{[-200,800]} p \wedge \Box \Diamond_{[0,1000]} p$. This creates a plateau of slightly higher progression time, which lasts until a state is reached where p is true. Then, the formula collapses back to its original form, at a slightly higher cost which results in a temporary peak in the diagram.

It should be clear from Figure 7.15 on the previous page that these performance results are not substantially affected by using other temporal intervals than 1000 ms. Formulas of the given type do not expand arbitrarily, and changing the sequence of false and true states provided to the progression algorithm merely changes the balance between the number of time-points where progression takes approximately 30, 40, and 47 μs. In other words, a timeout of one hour ($\Box \Diamond_{[0,3600000]} p$) is as efficiently handled as a timeout of one second.

7.10.2 Experiment: Always Not p Implies Eventually Always p

Let ϕ denote 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. In the second experiment, we progressed this formula through several different state sequences:

- **(true)** – as the inner formula only needs to be progressed when p is false, this is the best state sequence in term of performance.
- **(false*1,true*10)** – p is false during one sample and true during ten samples, after which the sequence repeats.

  As p is initially false, the formula is initially progressed to $(\Diamond_{[0,900]} \Box_{[0,999]} p) \wedge \phi$, where the first conjunct reflects the fact that 100 ms have already passed, leaving at most 900 ms until the required interval of length 1000 where p is true.

  Progressing through the next state sample, where p is true, results in $((\Box_{[0,899]} p) \vee \Diamond_{[0,800]} \Box_{[0,999]} p) \wedge \phi$. The first conjunct of the previous formula was $\Diamond_{[0,900]} \Box_{[0,999]} p$, and as p is now true, this can be satisfied in two ways: Either a sufficiently long interval where p is true begins now, extending 900 ms from the next state ($\Box_{[0,899]} p$), or it begins later, within 800 ms ($\Diamond_{[0,800]} \Box_{[0,999]} p$).

  After simplification, any further progression through a state where p is true will still result in a formula having the form $((\Box_{[0,i]} p) \vee \Diamond_{[0,j]} \Box_{[0,999]} p) \wedge \phi$, up to the point where p has been true for a sufficient period of time and the formula once again collapses to ϕ.

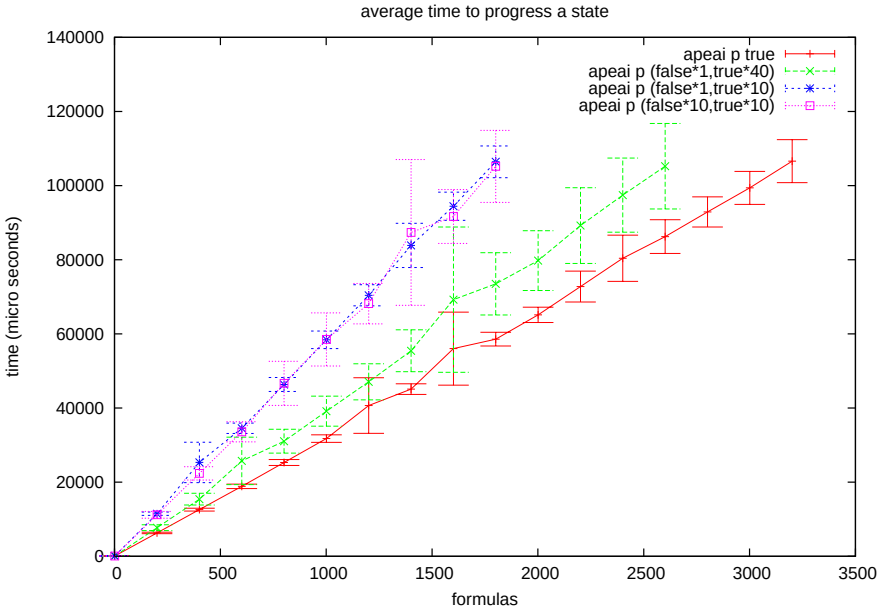


Figure 7.16: Testing: Always Not p Implies Eventually Always p (average progression time).

- **(false*1,true*40)** – p is false during one sample and true during forty samples, after which the sequence repeats.
- **(false*10,true*10)** – p is false during ten samples and true during ten samples, after which the sequence repeats.

The results shown in Figure 7.16 shows that 100 ms is sufficient for the progression of between 1500 and 3000 formulas of this form, depending on the state sequence.

Figure 7.17 on the next page shows the amount of time required to process 1000 instances of the formula $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} p$ through each individual state sample.

- In the case where p is always true, progression time is constant after the first progression step: The antecedent of the implication never holds, and the progression algorithm always returns the original formula. This is the best case, requiring approximately 32 μs for each progression step.
- Whenever p is false, the antecedent of the implication holds and the consequent $\Diamond_{[0,1000]} \Box_{[0,999]} p$ is progressed and conjoined to the current formula. Thus, if p alternates between being false for 1 sample and true for 10 samples, the formula expands only once. This results in a higher but constant "plateau" followed by a collapse back to the original formula.

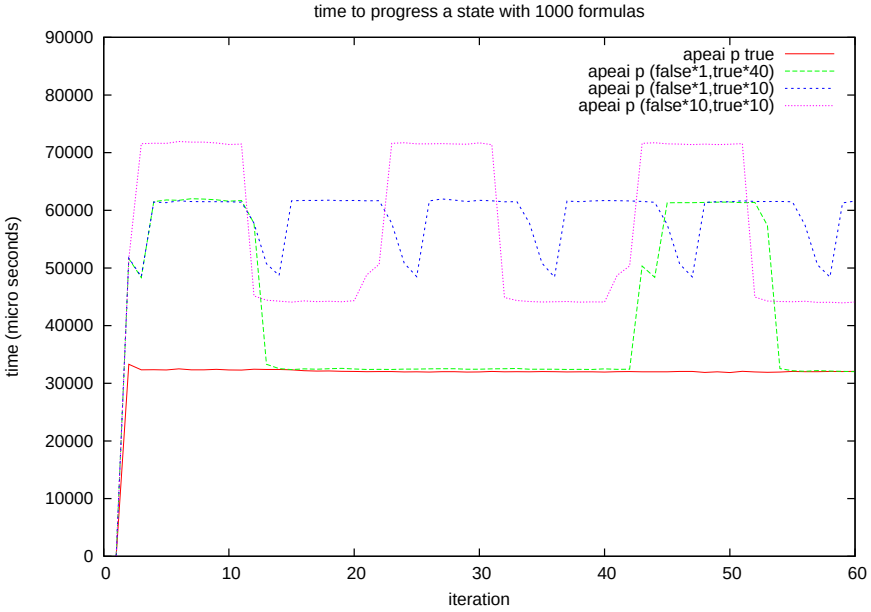


Figure 7.17: Testing: Always Not p Implies Eventually Always p (development over time).

- If p is true for more than 10 samples, progression eventually returns to the lowest plateau, as can be seen in the case where p alternates between being false for 1 sample and true for 40 samples.
- Finally, consider the case where p remains false for as long as possible (10 samples), after which it is true for the minimum period required (10 samples). In the first state where p is false, the antecedent of the implication holds and its consequent must be conjoined, just like before. However, this is now followed by another state where p is false. As the implication is triggered again, the formula temporarily expands even more. Though the formula immediately collapses due to subsumption, the additional processing leads to a high plateau where a single progression step may take as much as 72 μs.

Again, the figure indicates upper and lower bounds for progression time requirements: A single progression step may take between 32 and 72 μs, and altering permitted interval lengths only changes the range of permitted proportions between these time-points – in other words, the possible lengths of the plateaus. Regardless of interval lengths, the worst case cannot require more than 72 μs per iteration, which means that more than 1300 formulas can be progressed within a sample period of 100 ms.

7.11 Related Work

Many architectures that deal with both planning and execution focus entirely on recovery from detected problems by plan repair or replanning (Ambros-Ingerson and Steel, 1988; Finzi, Ingrand, and Muscettola, 2004; Haigh and Veloso, 1998; Lemai and Ingrand, 2004; Myers, 1999). These architectures usually assume that state variables are correctly updated and that plan operator implementations detect any possible failure in some unspecified manner, and thereby do not consider the full execution monitoring problem. A more elaborate and general approach is taken by Wilkins, Lee and Berry (2003) where a large set of different types of monitors are used in two different applications. However, in their approach monitors are procedurally encoded instead of using a declarative language.

In those architectures that do consider execution monitoring, it is often an intrinsic part of an execution component rather than integrated with the planner (Fernández and Simmons, 1998; Simmons and Apfelbaum, 1998). The most common approach uses a predictive model to determine what state a robot should be in, continuously comparing this to the current state as detected by sensors (Chien et al., 2000; Washington, Golden, and Bresina, 2000). This is a well-studied problem in control theory, where it is often called fault detection and isolation (FDI) (Gertler, 1998). Using models derived from first principles it is possible to detect faulty sensors and other components. The same approach has also been taken in planning, where the plan itself leads to a prediction of a state sequence that should occur (Fikes, 1971), as well as in path planning and robot navigation (Fernández and Simmons, 1998; Gat et al., 1990). For example, Gat et al. (1990) takes the output from a path planner and simulates the expected sensor readings the agent should receive when following the path. From these expectations, one derives for each sensor reading an interval of time within which the reading is expected to occur. While following the generated path, readings outside the expected interval cause the robot to switch to a recovery mode which attempts to handle the unintended situation.

Several significant weaknesses can be identified in this approach. The fact that one can detect a discrepancy between the current state and the predicted state does not necessarily mean that this discrepancy has a detrimental effect on the plan. Thus, one must take great care to distinguish essential deviations from unimportant ones, rendering the advantage of being able to automatically derive problems in execution from a predictive model considerably less significant. Similarly, that one can predict a fact does not necessarily mean that this fact must be monitored at all. Excessive monitoring may be unproblematic in a chemical processing plant where fixed sensors have been placed at suitable locations in advance and information gathering is essentially free, but does cause problems when monitoring costs are not negligible. Specifically, increasing the richness and fidelity of a domain model should not necessarily cause the costs for monitoring to increase.

The approach used by Fernández and Simmons (1998) focuses on undesirable behavior rather than expected situations, explicitly defining a set of hierarchically organized monitors corresponding to *symptoms* that can be detected by a robot.

Top level monitors cover many cases, but report problems with a large delay and provide little information about the cause of a problem. For example, a top level monitor may detect excessive action execution time, which is of little help if a problem occurs at the beginning of an action but covers any conceivable reason for the delay. Leaf monitors are more specific and provide more information, but may provide less coverage. While the idea of focusing on explicitly specified undesirable behavior avoids the problems discussed above, symptoms appear to be hardcoded rather than declaratively specified. As our approach can monitor both operator execution time, operator-specific constraints, and global constraints, the approach taken by Fernández and Simmons (1998) can be emulated in our system, including checking for special situations such as a robot getting stuck while spinning around.

While these approaches do cover some important aspects of the execution monitoring problem, they still generally fail to consider issues related to multiple agents, the information gathering problem, and the problem of incomplete information about the current state. Another major weakness is that only the current state is considered. Adapting ideas from model checking (Clarke, Grumberg, and Peled, 2000) to be able to talk about sequences of states, Ben Lamine and Kabanza (2002) expressed the desired properties of a system in a temporal logical formalism. Whereas model checking generally tests such properties against a system model, their execution monitor system tests them against an actual execution trace. Similar ideas have also been developed in the model checking community. There, the problem of checking whether a single execution trace of a system satisfies a property is called path model checking (Finkbeiner and Sipma, 2004; Markey and Raskin, 2006; Markey and Schnoebelen, 2003), or runtime verification if the evaluation is done incrementally as the trace develops (Barringer et al., 2004; Barringer, Rydeheard, and Havelund, 2008; Drusinsky, 2003; Rosu and Havelund, 2005; Thati and Rosu, 2005). These approaches are equivalent to progression of a formula and have been further extended to more expressive logics.

Though the work by Ben Lamine and Kabanza provided part of the inspiration for this chapter, it focuses on a reactive behavior-based architecture where the combined effects of a set of interactive behaviors is difficult to predict in advance. There, monitor formulas generally state global properties that cannot be monitored by internal behaviors, such as the fact that after three consecutive losses of communication, a given behavior must be active. A violation triggers an ad-hoc behavior that attempts to correct the problem. In comparison, our system is based on the use of planning, with an integrated domain description language. Formulas are not necessarily global, but can be operator-specific. Our approach provides a selective mechanism for extracting monitor formulas through automated plan analysis and supports recovery through replanning. DyKnow also gives us the possibility to provide attention focused state descriptions, where the execution monitor contextually subscribes only to those state variables that are required for progressing the currently active monitor formulas. Additionally, state sampling rates can be decided individually for each monitor formula. Combining this with the use of operator-specific monitor formulas that are only active when specific tasks are be-

ing performed ensures that state variables are only computed when stricly required, ensuring minimal usage of resources including sensors and computational power. This can be particularly beneficial in the case of state variables requiring image processing or other complex operations.

See Pettersson (2005) for an overview of systems related to execution monitoring.

7.12 Conclusions and Future Work

In this chapter, we have presented an architectural framework for planning and execution monitoring where conditions to be monitored are specified as formulas in a temporal logic. A key point in this architecture is the use of the same logic formalism, TAL, for both planning and monitoring. This allows a higher degree of integration between these two important topics in several respects, including the fact that conditions to be monitored can be attached directly to specific actions in a plan and that a plan can be analyzed to automatically extract conditions to be monitored.

Another key point is that formulas are evaluated on a sequence of states corresponding to an approximation of the actual development of the world. To construct this state sequence from sensors and other information sources DyKnow was extended with a state synchronization mechanism. The generated state sequences can be seen as partial logical models of the world over which formulas are evaluated.

The framework we presented has been implemented and integrated into our unmanned aerial vehicle platforms. The system has been deployed and used in actual missions with the UAVs in a smaller urban environment. Empirical testing has taken several different shapes during the course of the project. While an early version of this system was tested on board the UAV, the full logistics scenario cannot be tested until a winch and an electromagnet are available, together with suitably prepared boxes and carriers. These devices are under development and will be ready for use in the near future. Until then, only monitor formulas related to pure flight have been tested on the physical system, and ironically (or fortunately) the UAV system has proved quite stable and few opportunities for failure detection have presented themselves. Therefore, the most intensive tests have been performed through simulation.

In terms of testing their adequacy for detecting failures, monitor formulas have been tested through intentional fault injection, where one may, for example, simulate dropping a box or failing to take off. Surprisingly often, formulas have also been tested through unintentional failures. For example, when ordered to detach a box, the simulator does not simply place them at the designated coordinates; instead, it turns off the simulated electromagnet, after which the box falls from its current altitude towards the ground. This, in turn, may make the box bounce or roll away from its intended coordinates. Taken together, the simulated environment is most likely as good at testing a wide variety of failures as the physical UAV system, and certainly more efficient.

Finally, the computational adequacy of the system is also important. What matters in this area is not the analytically derived worst case temporal complexity for the most complex monitor formulas, but the actual worst case and average case performance for those formulas that are actually useful and relevant for a domain. Extensive testing has therefore been done by running sets of typical monitor formulas in parallel using constructed best-case and worst-case state sequences. Results indicate that while there is a certain cost associated with generating state sequences, the additional cost for each new formula is very low given the typical structure of monitor formulas. Thus, given that the appropriate sensor values are available in DyKnow, execution monitoring does not require significant additional resources, even if large numbers of monitor formulas are active concurrently.

An interesting topic for future research is that of autonomously determining the state that results from a failure. For example, given that a UAV detects that it dropped a box, where did that box end up? In the first stage, a mixed initiative approach may be appropriate, where the UAV signals a failure to a human operator which helps the UAV find the box, possibly aided by the camera on board the UAV. Ideally, the UAV should be able to find the box completely autonomously; for example, by flying a regular scanning pattern and using its vision subsystem, a laser scanner, or other remote sensing equipment to find likely candidates. Here, information about previously performed actions can potentially be of value in determining the most likely location of a box, as well as information from geographic information systems and other sources of data regarding the environment.

Based on a great deal of experience with our UAV systems, it is our strong belief that using logics as the basis for deliberative functionalities such as planning and monitoring the expected effects of plans in the environment simplifies the development of complex intelligent autonomous systems such as UAVs. Temporal Action Logic and its tense formula subset are highly expressive languages which are well suited for describing the UAV domain and for expressing the monitoring conditions we are interested in. Therefore we believe this approach provides a viable path towards even more sophisticated and capable autonomous UAV applications.

Chapter 8

Integrating Object and Chronicle Recognition

8.1 Introduction

Many applications of autonomous aerial and ground vehicles involve surveillance and monitoring where it is crucial to recognize *objects* existing in the environment and *events* related to these objects. For example, a UAV monitoring traffic must be able to determine whether a blob detected by image processing is likely to be a vehicle or a building. It must also 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. These are prime examples of the type of tasks knowledge processing middleware are intended to facilitate.

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.

One formalism to express complex events is the chronicle formalism (Ghallab, 1996) which represents and detects complex events described in terms of temporally constrained events. We have successfully used chronicles to describe traffic behavior in a traffic monitoring application. Instances of these chronicles are detected by C.R.S., an implementation of the chronicle recognition algorithm developed by France Telecom (CRS website). The chronicle formalism is described in Sections 8.2 – 8.4.

In a similar manner objects can either be observed or inferred from other observations. Which types of observations are available is very much dependent on the platforms and sensors being used. If a camera is mounted in the front of a car then it will have a very limited view of the surroundings and will most likely only

see parts of the cars around it but with quite a lot of details, such as the brand or the license plate number. On the other hand, a camera mounted underneath a helicopter will have a quite good overview of an area but with less detailed information about the cars. In both cases it is important to be able to describe different classes of objects and detect individuals from these classes.

What features are associated with an object depends on the classification: A vehicle has a current velocity, but a building does not. Classification must also be flexible over time. For example, if a blob previously thought to be a building begins to move, it should be reclassified.

Supporting this requires a flexible framework for object classification and reclassification. We have developed an approach where objects are *incrementally* classified as belonging to increasingly strict classes in a hierarchy of types. To provide the required flexibility, a separate object structure is created for each type an object is believed to belong to. For example, any object detected by image processing may be represented as a *vision object* structure. If the vision object satisfies the requirements for being a vehicle, a *vehicle object* structure is instantiated and a link is created between the vision object structure and the vehicle object structure. The vehicle object structure can then include additional features not available in arbitrary vision objects.

Note that each link and each object structure is only seen as a hypothesis about the class and the identity of an object. These hypotheses are continually monitored using the same approach as for execution monitoring. Each hypothesis is associated with a metric temporal logical formula which is incrementally evaluated using progression. If a monitor formula is violated the hypothesis is removed. For example, if an object believed to be a vehicle violates the expected behavior, the link to the associated vision object structure is removed, but the vision and vehicle object structures themselves remain.

A set of linked object structures forms an *object linkage structure*. This approach is described in Section 8.5.

One problem that has to be dealt with is connecting a symbolic representation of a car to a stream of sensor data collected by the UAV in such a way that the symbol actually represents the car in the world. This is called the anchoring problem (Coradeschi and Saffiotti, 2003). The goal is to associate symbols with sensor data in such a way that a symbol and its associated sensor data refers to the same object. By creating and maintaining this association the symbol can be said to be *anchored* or more generally *grounded*. How we use object linkage structures to anchor objects is described in Section 8.6.

An equally important functionality is to be able to integrate sensors, such as cameras, and processing of sensor data, such as image processing, with the detection of objects and complex events. This integration is a typical use of knowledge processing middleware. To give a concrete example of how this can be done using DyKnow, we present in Section 8.7 an implemented and tested traffic monitoring application which integrates both object and chronicle recognition.

8.2 The Chronicle Formalism

A chronicle is a description of a complex event representing a generic scenario whose instances we would like to recognize. A chronicle is represented as a set of events and a set of metric temporal constraints on these events (Ghallab, 1996). In this context, an event is often defined as a change in the value of a feature.

The chronicle recognition algorithm takes a stream of time-stamped event occurrences and finds all matching chronicle instances online. To do this, the algorithm keeps track of all possible developments (Ghallab calls this prediction) in an efficient manner by using temporal constraint networks (Dechter, Meiri, and Pearl, 1991). A chronicle instance is matched if instances of all the events in the chronicle model are present in the stream and the time-stamps of the event instances satisfy the temporal constraints. Recognized instances of a chronicle can be used as events in another chronicle.

Example 8.2.1 (Overtake) A typical traffic situation that can be expressed as a chronicle is an overtake. A chronicle where car A overtakes car B can be described as a chain of events where a car A is first behind, then left of, and finally in front of car B while both cars are on the same road during the entire overtake and with the constraint that the whole overtake should not take more than 1 minute. □

Example 8.2.2 (Reckless overtake) A traffic situation which might be more interesting to detect is reckless overtakes. There are several variations of the overtake sequence of events which could be seen as a reckless overtake. One classical example is an overtake on the wrong side of a car. In a country with right-hand traffic, an overtake should be made on the left side of the car and not the right. Another example is when the overtaking car does not have enough distance to a meeting car. A third example is when a car begins and interrupts an overtake over and over again. □

Another application area for chronicle recognition, apart from the traffic monitoring scenario described in this chapter, is in the surveillance of dynamic systems. It has for example been used with success to monitor gas turbines (Aguilar et al., 1994) and telecommunication networks (Bibas et al., 1996).

The following sections give a high level description of the chronicle formalism as presented in Ghallab (1996), Dousson (2002), and the CRS website. We start with presenting the chronicle language (Section 8.3) and then the online recognition algorithm used to detect instances (Section 8.4).

8.3 The Chronicle Language

In Example 8.2.1 we presented an overtake as a typical example of a complex event that can be expressed as a chronicle. In this section we introduce the chronicle language, as it is defined by C.R.S. (CRS website), which can be used to describe such complex events. Let us start with an example.

Example 8.3.1 (Chronicle) To use chronicle recognition to detect overtakes we need to translate the natural language description of an overtake given in Example 8.2.1 to a formal chronicle. To model overtakes four features are used: behind, left, in_front, and road. The first three features represent qualitative spatial relations between pairs of cars and the last represents which road a car is on.

Given these four features it is possible to translate the description of an overtake to a chronicle using changes in these features as primitive events:

chronicle overtake[*?car1*,*?car2*]
{
  event(behind[*?car1*, *?car2*]:(*?*, true), t1)
  event(left[*?car1*, *?car2*]:(false, true), t2)
  event(in_front[*?car1*, *?car2*]:(false, true), t3)
  event(road[*?car1*]:(*?*, *?road*), t4)
  noevent(road[*?car1*]:(*?road*, *?*), (t4, t3))
  event(road[*?car2*]:(*?*, *?road*), t5)
  noevent(road[*?car2*]:(*?road*, *?*), (t5, t3))
  t4 ≤ t1; t5 ≤ t1; t1 ≤ t2
  t3 - t2 **in** [0, 60000]
}

The first three rows state that an overtake consists of three primitive events, where a car is first behind, then left of, and finally in front of another car. The next four rows state that the two cars must be on the same road during the whole overtake. Specifically, cars 1 and 2 must enter the same road ?road at times t4 and t5, respectively, after which there must be no events where the cars leave this road before time t3. The last two rows state the temporal constraints on the occurrence times of the primitive events. The time unit is milliseconds. □

We will now present the language in some detail. A formal grammar is presented in Section 8.3.5 on page 153.

8.3.1 Symbol

Symbols are used for naming chronicle models, event types, time-points, variables, and their values. A symbol may be any string of alphabetical characters (a-z, A-Z), the underscore (_), and digits (0-9) with the restriction that the first character must not be a digit. Some example symbols: overtake and car2.

Some symbols can not be used as names since they are keywords. The following are all the reserved keywords: **alias and attribute chronicle constraint delay domain event hold in init instant message noevent not occurs oo -oo or recognized send variable when ?value**.

Variable

Variables are symbols prefixed by a question mark '*?*' or a star '*'. A variable whose name begins with a question mark is constrained to taking on the same value

throughout a chronicle, while a variable whose name begins with a star can take on different values on each occurrence and therefore serves as a type of placeholder where the precise value of an argument is not relevant.

Time Constant and Time Interval

All time constants are expressed as integers.

Let t_1 and t_2 be two time constants, and **oo** and **-oo** two keywords representing positive and negative infinity, respectively. Then, the following are the allowed *time intervals*:

- $[t_1, t_2]$ corresponds to $\{t \in \mathbb{Z} \mid t_1 \leq t \leq t_2\}$,
- $[t_1, \textbf{oo}[$ corresponds to $\{t \in \mathbb{Z} \mid t_1 \leq t\}$,
- $]\textbf{-oo}, t_1]$ corresponds to $\{t \in \mathbb{Z} \mid t \leq t_1\}$, and
- $]\textbf{-oo}, \textbf{oo}[$ corresponds to $\mathbb{Z}$.

Example 8.3.2 (Time constants and time intervals)
The following expressions are examples of valid time constants and time intervals:

- 123456,
- [10, 122], and
-]**-oo**, **oo**[.

□

Temporal Expression

A *temporal expression* is an expression on one of the following forms:

- $t_1 \otimes t_2$, where t_1 and t_2 are time constants and $\otimes$ is one of the binary operators + (addition) and − (subtraction),
- $i_1 \otimes i_2$, where i_1 and i_2 are time intervals and $\otimes$ is one of the binary operators & (intersection) and | (union), and
- $i_1 \otimes t_1$, where i_1 is a time interval, t_1 is a time constant, and $\otimes$ is one of the binary operators + (addition) and − (subtraction), meaning that the given time constant is added or subtracted from every finite time constant occurring in the time interval.

Example 8.3.3 (Temporal expressions)
The following expressions are examples of valid temporal expressions:

- [−12, 100] + 22 which is equal to [10, 122],
- [1300, **oo**[& [0, 10000] which is equal to [1300, 10000], and
- [10, 122] | [50, 100] which is equal to [10, 122].

□

Domain

A domain consists of a set of values that a feature can take on and is specified using ordinary set notation as a comma-separated list of symbols surrounded by braces. Three binary operations are defined on domains: union (| or +), intersection (&), and subtraction (−). Additionally, the unary prefix complement operator (∼) returns the complement of a domain relative to the set of all values occurring in a chronicle specification.

As will be seen later, domains can be used inline in parameter specifications. A global domain can also be declared and given a name using a declaration of the following form: **domain** name = {value1, . . . , valueN}.

Example 8.3.4 (Domains)
The following expressions are examples of valid domains:

- **domain** Color = {red, blue, green},
- **domain** Car = {car3} | {car2} | {car1} which is equal to {car1, car2, car3}, and
- **domain** All = ∼ {} which is the domain of all symbols.

□

8.3.2 Attribute and Message

Before defining events and chronicle models, the meaning carried by these events must be defined. In the chronicle formalism, there are two kinds of entities from which events can be derived:

- *attributes* which have values over time and where events correspond to changes of this value, and
- *messages* which are instantaneous events with no duration, where a message indicates the occurrence of such an event.

An attribute is similar to a feature whose value may change over time. In DyKnow the value of an attribute over time is represented by a fluent stream, while in the chronicle formalism, it is represented by a sequence of events where each event represents a change in the value. A message corresponds to an event which does not necessarily represents a change in the value of an attribute.

Message

A message is defined by a block which starts with the keyword **message** followed by the name and parameters of the message. The block defines a message type which can be received by the recognition process.

Each parameter has a domain, which may be declared in the body of the message. The default domain is the domain of all symbols. A parameter can either have an explicit domain or refer to one of the global domains.

Example 8.3.5 (Message) A message can for example be used to represent that a car is in a crossing:

```
message in_crossing[?car, ?crossing]
{
    ?car in Car
    ?crossing in Crossing
}
```

□

Attribute

An attribute is defined by a block which starts with the keyword **attribute** followed by the name and parameters of the attribute. The block defines an attribute which has a value that can change over time. Its value domain is represented by a predefined variable called **?value**. As for messages, the domains for the parameters may be defined.

Example 8.3.6 (Attribute) An attribute on_road representing the boolean feature of a car either being on a road or not can be defined as:

```
attribute on_road[?car]
{
    ?value in {true, false}
    ?car in Car
}
```

□

8.3.3 Time Constraint

Time constraints are defined between time-points, also called instants, which are the temporal variables of a chronicle. These temporal variables are not prefixed with a question mark or a star. Instants are implicitly declared when they are first used. However, it is also possible to explicitly declare instants by the keyword **instant**. For example: **instant** t1, t2.

Two instants are always defined for a chronicle, start and end. These are instants with the constraint that start $\leq t \leq$ end for any instant t of the chronicle model. The consistency of all the constraints of a chronicle model is checked at compile time.

Atomic Constraint

An atomic constraint is a constraint on the form $t_1 \otimes t_2$ or t_1**in** I, where t_1 and t_2 are temporal expressions, $\otimes$ is an operator in $\{<, <=, =, >=, >\}$, and I is a time interval. For representational convenience this syntax is extended to allow chained expressions such as $t_1 \otimes_1 t_2 \otimes_2 t_3$, meaning that $t_1 \otimes_1 t_2$ and $t_2 \otimes_2 t_3$.

Example 8.3.7 (Atomic Constraints)
The following constraints are examples of atomic constraints:

- t1 < t2 <= t3 = t4,
- t3 − t2 **in** $[0, 250]$,
- t4 − t1 < 100, which is equal to t4 − t1 **in**]**-oo**, 100[, and
- 10 <= t5 − t1 <= 30.

□

Using the start and end instants it is easy to define the maximum duration of a chronicle like this: end − start <= 10.

Complex Constraint

A complex constraint is defined by a block which starts with the keyword **constraint** followed by the name and parameters of the constraint. The parameters must be time instants. The block defines a complex temporal constraint as a conjunction of atomic constraints. The new constraint can then be used in chronicles to avoid having to duplicate complex constraints multiple times. This feature is purely syntactic.

Example 8.3.8 (Complex Constraint)
A sequence of 3 instants with a maximum duration of 5 time units can be defined as follows:

```
constraint mySequence(t1, t2, t3)
{
    t1 <= t2 <= t3
    t3 − t1 <= 5
}
```

The constraint can then be used in a chronicle like this:

```
instant begin, middle, finish
mySequence(begin, middle, finish)
```

□

8.3.4 Chronicle Model

A chronicle model is defined by a block which starts with the keyword **chronicle** followed by the name and parameters of the chronicle. The block defines one chronicle model whose instances may be recognized by the online chronicle recognition engine.

Example 8.3.9 (Chronicle model)

chronicle overtake[*?car1*,*?car2*]
{
    ...
}

□

A chronicle model may include local variables, event occurrences, and temporal constraints.

Local Variable

If an expression of the form *?var* or **var* is used within a chronicle without being declared, it is implicitly declared as a local variable in that chronicle. It is also possible to declare local variables explicitly, either to improve clarity or to define a domain to which the variable should belong. A variable definition has the following form, where each var_i is a variable name beginning with '*?*' or '*':

- **variable** *var1* [, *var2*, ...]
- **variable** *var1* [, *var2*, ...] **in** *domain*

Example 8.3.10 (Local Variables)
The following statements are examples of valid local variable declarations:

- **variable** *?y*, **z*
- **variable** *?c* **in** *Car*

□

Occurs Predicate

An occurs predicate declares that a specific number of event instances must occur within a specific interval for an instance of the chronicle to be recognized. All the occurs predicates in a chronicle model must be matched in order for a chronicle instance to be detected. The syntax is **occurs**$(n_1, n_2, TYPE, (t_1, t_2))$, where $0 \leq n_1 \leq n_2$ are integers, $TYPE$ is an event type, and $t_1 \leq t_2$ are time-points. The meaning of the declaration is that at least n_1 but not more than n_2 events of the type $TYPE$ should occur in the interval $[t_1, t_2[$. To represent that any number of event instance may occur n_2 can be replaced by the keyword **oo**.

The type of an event depends on whether it is an instance of a message or a value change of an attribute. In the first case, the event type has the form $n[a_1, \ldots, a_n]$, where n is the name of the message and $a_1, \ldots, a_n$ are its arguments, which may be variables or values. In the case of a change in the value of an attribute, the event type has the form $n[a_1, \ldots, a_n] : (v, v')$, where n is the name of the attribute, $a_1, \ldots, a_n$ are the arguments of the attribute, v is the previous value, and v' is the current value.

Example 8.3.11 (Occurs predicate)

- **occurs**(1, 1, in_crossing[car1, *?c*], (t1, t2)) states that car1 should be in a crossing *?c* exactly once in the interval [t1, t2[.
- **occurs**(1, 5, in_crossing[*?car2*, *?*], (t3, t4)) states that a car, denoted by the variable *?car2*, should enter the same crossing between 1 and 5 times in the interval [t3, t4[.
- **occurs**(1, **oo**, on_road[*?car1*] : (*?*, false), (t5, t6)) states that the attribute on_road for a car (denoted by *?car1*) should change values to false (from any value) at least once in the interval [t5, t6[.

□

Notice that *?* can be used instead of a variable to represent that the actual value matched during recognition is not important. Each *?* represents a different variable.

Counting events with different arguments in the same occurs predicate is possible by the use of universal variables, which are variables prefixed by a star (*) instead of a question mark (*?*).

Example 8.3.12 (Occurs predicate, cont.)

- **occurs**(3, 3, in_crossing[*?car*, *?c*], (t1, t2)) states that the same car should be in the same crossing exactly three time during the interval [t1, t2[.
- **occurs**(3, 3, in_crossing[**car*, *?c*], (t1, t2)) states that it might be different cars which should be in the same crossing exactly three times during the interval [t1, t2[.

□

For compatibility and convenience, it is possible to use the predicates **event** (meaning one or more events) and **noevent** (meaning zero events) in a chronicle model. The definition of these predicates are:

event$(TYPE, t)$ = **occurs**$(1, \mathbf{oo}, TYPE, (t, t+1))$
noevent$(TYPE, (t_1, t_2))$ = **occurs**$(0, 0, TYPE, (t_1, t_2))$

To declare a context assertion stating that an attribute p has a certain value v over a certain interval (t_1, t_2) an **event** predicate, a **noevent** predicate, and a temporal constraint is used:

event(p : (*?*, v), t0)
noevent(p : (v, *?*), (t0 + 1, t2))
t0 <= t1

A chronicle model is a conjunction of **occurs** predicates and temporal constraints representing the spatio-temporal pattern of a complex event.

Example 8.3.13 (Chronicle model, cont.) We can now describe an overtake as a chronicle according to the definition in Example 8.2.1.

chronicle overtake[*?car1*,*?car2*]
{
event(behind[*?car1*, *?car2*]:(*?*, true), t1)
event(left[*?car1*, *?car2*]:(false, true), t2)
event(in_front[*?car1*, *?car2*]:(false, true), t3)
event(road[*?car1*]:(*?*, *?road*), t4)
noevent(road[*?car1*]:(*?road*, *?*), (t4, t3))
event(road[*?car2*]:(*?*, *?road*), t5)
noevent(road[*?car2*]:(*?road*, *?*), (t5, t3))
t4 ≤ t1; t5 ≤ t1
t1 ≤ t2
t3 - t2 **in** [0, 60000]
}

□

8.3.5 Grammar

The following is the grammar of C.R.S. according to the CRS website.

```
      ChronicleFile ::= (  DomainDefinition
                        |  ConstraintModel
                        |  TimeConstraintGraph
                        |  Attribute
                        |  Message
                        |  Chronicle  )*
   DomainDefinition ::= domain  <ID>  '='  Domain
             Domain ::= IntersectedDomain
                        (  (  '|'  Domain  |  '\'  Domain  )  )?
 IntersectedDomain ::= BasicDomain
                        (  '&'  IntersectedDomain  )?
       BasicDomain ::= '('  Domain  ')'
                     | '~'  BasicDomain
                     | '{'  (  <ID>  (  ','  <ID>  )*  )?  '}'
                     | <ID>
    ConstraintModel ::= constraint  Signature
                        '{'  (  Constraint  )*  '}'
         Constraint ::= ConstraintDisjunct
                        (  '=>'  ConstraintDisjunct
                        |  '<=>'  ConstraintDisjunct  )?
                     | if  ConstraintDisjunct
                        then  ConstraintDisjunct
                        (  else  ConstraintDisjunct  )?
ConstraintDisjunct ::= ConstraintConjunct
                        (  or  ConstraintDisjunct  )?
ConstraintConjunct ::= AtomicConstraint
                        (  and  ConstraintConjunct  )?
```

```
    AtomicConstraint ::= Variable
                         ( ( not )? in Domain
                           | ( '==' | '!=' ) Parameter )
                       | not AtomicConstraint
                       | '(' ConstraintDisjunct ')'
           Parameter ::= ( Variable | '?' | '*' )
                       | <ID>
            Variable ::= ( '?' | '*' ) <ID>
 TimeConstraintGraph ::= time constraint <ID>
                         ( TemporalParameters )?
                         '{' ( TimeConstraint )* '}'
      TimeConstraint ::= TimePoint
                         SymbolicInstantSeq
                       | <ID> ( - <ID> )?
                         ( in TimeInterval
                         | ( '<' | '>'
                           | '<=' | '>=' ) TimeValue )
                       | <ID> TemporalIndexes
  SymbolicInstantSeq ::= '=' TimePoint
                         ( SymbolicInstantSeq )?
                       | '<' TimePoint
                         ( IncreasingInstantSeq )?
                       | '>' TimePoint
                         ( DecreasingInstantSeq )?
                       | '<=' TimePoint
                         ( IncreasingInstantSeq )?
                       | '>=' TimePoint
                         ( DecreasingInstantSeq )?
IncreasingInstantSeq ::= ( '=' TimePoint
                         | '<' TimePoint
                         | '<=' TimePoint )
                         ( IncreasingInstantSeq )?
DecreasingInstantSeq ::= ( '=' TimePoint
                         | '>' TimePoint
                         | '>=' TimePoint )
                         ( DecreasingInstantSeq )?
  TemporalParameters ::= '(' ( <ID> ( ',' <ID> )* )? ')'
     TemporalIndexes ::= '(' ( TimePoint
                         ( ',' TimePoint )* )? ')'
           TimeValue ::= ( '+' )? <INTEGER>
                       | '-' <INTEGER>
           TimePoint ::= <ID>
                         ( '+' ( <INTEGER>
                           | TimeInterval )
                         | '-' ( <INTEGER>
```

```
                          | TimeInterval ) )?
       TimeInterval ::= AtomicTimeInterval
                        ( ( '+' TimeInterval
                          | '−' TimeInterval ) )?
AtomicTimeInterval ::= ( '[' TimeValue
                       | ']' -oo ) ','
                       ( oo '[' | TimeValue ']' )
                     | '−' AtomicTimeInterval
          Attribute ::= attribute Signature
                        ( '{' ( Constraint )* '}' )?
            Message ::= message Signature
                        ( '{' ( Constraint )* '}' )?
          Chronicle ::= chronicle Signature
                        ( TemporalParameters )?
                        '{' ( ChronicleStatement )* '}'
ChronicleStatement ::= Predicate
                     | Constraint
                     | TimeConstraint
                     | chronicle Signature
                       ( TemporalLabeledIndexes )?
          Signature ::= <ID> ( Parameters )?
         Parameters ::= '[' ( Parameter
                        ( ',' Parameter )* )? ']'
          Predicate ::= event '(' Type ',' TimePoint ')'
                     | noevent '(' Type ','
                       '(' TimePoint ','
                       TimePoint ')' ')'
                     | occurs '(' <INTEGER> ','
                       ( <INTEGER> | oo ) ','
                       Type ',' '(' TimePoint ','
                       TimePoint ')' ')'
                Type ::= Signature
                       ( ':' '(' Parameter ','
                       Parameter ')' )?
```

8.4 On-Line Recognition

Given a chronicle model and a stream of events, the chronicle recognition system should as soon as possible detect any instance of the given chronicle model that occurs in the event stream. A complete match for a chronicle model associates every parameterized event in the chronicle with a corresponding concrete event in the event stream, in a way that is consistent with the chronicle constraints. A partial match is similar, but only requires a non-empty subset of the events required by a chronicle to be matched in a way that is consistent with the constraints.

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 (Nebel and Burckert, 1995; Vilain and Kautz, 1986) 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.

If a chronicle definition only contains **event** predicates and does not contain **noevent** or **occurs** predicates, either directly or indirectly through inclusion of other chronicle definitions, then it cannot depend on the non-existence of events or on the exact number of times an event occurs. To detect instances of such a chronicle it is first compiled into a simple temporal constraint network (Dechter, Meiri, and Pearl, 1991) where each event is a node and each temporal constraint is an edge between two nodes. A temporal constraint network is a directed acyclic graph where the nodes represent occurrences of events and each edge is associated with a set of disjunctive temporal constraints on the time of occurrence of the two connected events. A simple temporal constraint network is a temporal constraint network where each edge is only associated with a single temporal constraint.

A temporal constraint network corresponds to a temporal constraint satisfaction problem (TCSP) where a set of variables, one for each node, should be assigned a time-point in such a way that the binary constraints defined by the edges are satisfied. If it possible to find such an assignment then the temporal constraint network is consistent. Determining if a TCSP is consistent is NP-complete. However, if we restrict the problem to only allow a single temporal constraint on each of the edges in the temporal constraint network then we get a simple temporal constraint network whose corresponding simple TCSP can be solved in polynomial time (Dechter, Meiri, and Pearl, 1991).

When compiling a chronicle model, a complete temporal constraint network is created by propagating all the constraints in the model using an incremental path consistency algorithm (Ghallab, 1996). The result of this propagation is the least constrained complete graph equivalent to the constraints in the chronicle model. The compilation of a chronicle model also checks to make sure that the constraints are consistent. Since no disjunctive metric constraints are allowed the resulting network will be simple.

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 (Ghallab, 1996). 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.

For example, assume we have a chronicle model that requires three events A, B, and C to be matched together with a temporal constraint stating that C must occur

at most 5 time units after B. Also assume we have an event stream containing four event occurrences (4, A), (5, B), (8, B), and (12, C), where the integer is the time of the occurrence. The following will happen when the events occur:

- First, there is only the original instance [] where no events have been matched so far.
- (4, A) occurs. The only existing chronicle instance requires a match for the event A, but modifying it to add the fact that A is matched at time 4 would make it impossible to match A at a later time. Therefore, the existing chronicle is left intact and a copy is made in which A is matched at time 4. The new copy is consistent, and the result is two partial chronicles: [] and [(4, A)].
- (5,B) occurs. Both of the existing partial chronicle instances can be extended consistently with this event, resulting in four partial chronicles: [], [(4, A)], [(5, B)], and [(4, A), (5, B)].
- (8, B) occurs. Since chronicles are not destructively modified, we retain the ability to match this event against the original instance as well as the partial instance [(4, A)]. The remaining two chronicle instances already contain matches for B and cannot be extended with the new event. This results in six partial chronicles: [], [(4, A)], [(5, B)], [(8, B)], [(4, A), (5, B)], and [(4, A), (8, B)].
- The clock turns to 11. Recall that the chronicle constrains event C to happen within 5 time units after B. This is still possible for the partial instances where B is not matched or where B was matched at time 8. However, the partial instances where B was matched at time 5 can no longer be extended to complete instances and can be removed. This results in four active partial chronicles: [], [(4, A)], [(8, B)], and [(4, A), (8, B)].
- (12, C) occurs. This event can be added to all of the partial instances and results in one completely instantiated chronicle. Result: [], [(4, A)], [(8, B)], [(12, C)], [(4, A), (12, C)], [(8, B), (12, C)], [(4, A), (8, B)], and [(4, A), (8, B), (12, C)].

Translating a chronicle to a simple temporal constraint network only works for those chronicle models which consist of **event** predicates and temporal constraints. If a chronicle model contains **noevent** predicates or **occurs** predicates then a more elaborate evaluation mechanism is required (Dousson, 2002).

The **occurs**$(n_1, n_2, e, (t_1, t_2))$ predicate can be handled by translating it to a node in the temporal constraint network which is extended with an event counter n. For each event that matches the node and satisfies the constraints, the counter n is increased. A tree of partially instantiated chronicles is used to efficiently match incoming events and to keep track of the current state of the recognition process.

If $n < n_1$, then one has not yet received a sufficient number of events of the specified type. If additionally the current time is greater than t_2, then the deadline

has passed and the current chronicle instance cannot be matched regardless of what happens in the future.

If $n > n_2$, then too many events of the specified type have been received and the chronicle instance cannot be matched.

If $n_1 \leq n \leq n_2$ for all event counters then the chronicle instance is said to be ready to be recognized. However, if the current time is less than t_2 for some **occurs** predicate, then there is still a possibility that the upper limits will be exceeded before the deadline. To handle this each partial instance must also be updated when the clock is updated. The occurs predicate will be satisfied as soon as the interval (t_1, t_2) has passed and event counters are in their specified intervals.

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. The recognition process must be initialized with an attribute change event for each attribute with an occurrence date equal to $-\infty$.

8.5 Object Recognition and Tracking

Recognizing, tracking, and reasoning about various types of objects is an important problem when bridging the gap between sensing and reasoning. Therefore, knowledge processing middleware should provide support for managing these tasks. Throughout this section we will use traffic monitoring as a motivating example. In this problem domain, we are primarily interested in being able to recognize and track cars and other vehicles using thermal and color cameras mounted on a UAV.

For a UAV to be able to recognize a car it has to take a picture of the area where the car is and correctly identify that it actually is a car. To further be able to track the car as it moves through a road network the UAV has to continually take pictures in order to keep its information up to date. To do this the UAV has to be able to determine if a car found in a picture is actually the same car as it has seen before or if it is a previously unseen car.

Ideally, we would like to know the exact position, speed, and other attributes of every car in a monitored area at every time-point. However, the UAV may be given a large area to monitor, and since the cameras only can cover a small area at any time we can not expect to see all the cars in the area all the time. A UAV can therefore only track a subset of the cars, namely those cars that are observable by its cameras. Noise, lack of detail, and ambiguities in the images makes recognizing and tracking cars a challenging task.

In this section we will present an extension of DyKnow called *object linkage structures* which can be used to incrementally classify potential cars found by an image processing system and anchor those that are classified as cars to symbolic identifiers in order to support car recognition and tracking. How these object linkage structures can be used to anchor object symbols is described in Section 8.6.

8.5.1 Object

Most knowledge processing applications are possible to describe in terms of fluent streams approximating features. However, many of the features and their associated fluent streams are related to the same *object*. These applications could therefore be simplified by introducing support for processing objects consisting of bundles of related features. This is especially true if the set of objects is not fixed, but may change at run-time.

We consider an object to be an entity which has attributes and can have relations to other objects. Example objects in the traffic domain are cars, buildings, and roads. The attributes and the relations of an object are represented by features whose values over time are approximated using fluent streams.

To get all the information about a single object a client has to subscribe to each fluent stream approximating an attribute of the object. If a client needs to synchronize the information, then the state extraction mechanism of Chapter 7 can be used. However, in many cases a single process approximates more than one attribute of an object, maybe even all the attributes of an object. An improvement would be to allow a single process to approximate object states, i.e. snapshots of a collection of attributes about a single object. A knowledge process creating a stream of such object states would conceptually be seen as creating a bundle of related streams and an implementation could very well allow each stream to be accessed individually.

To manage such collections of streams we introduce *object states* as a specialization of a state as defined in Section 7.8. An object state represents a snapshot of the state of a particular object. Formally an object state is a state where the first value is the identifier of the object. A stream of object states is called an *object state stream*.

Throughout the rest of the chapter, we will assume the use of the knowledge processing domain $D = \langle O, T, P \rangle$, where $\mathbb{R} \subseteq P$ to allow real valued attributes.

Definition 8.5.1 (Object state) An *object state* in a knowledge processing domain $D = \langle O, T, P \rangle$ is a state $\langle o, v_1, \ldots, v_n \rangle$, where $o \in O$ is an object, and $v_1, \ldots, v_n \in V_D$ are the current values of the attributes of the object. □

Definition 8.5.2 (Object state stream) An *object state stream* in a knowledge processing domain D is a state stream in D where each state is an object state. □

Example 8.5.1 (Object) To model the traffic domain we introduce car objects to represent cars. Each car object would have its own object identifier, such as car72, bil, or a. If cars have the attributes position, speed, and size, then a snapshot of car72 could be represented by an object state $\langle$car72, $\langle 201, 75, 88 \rangle$, $\langle 20, 0, 0 \rangle$, $\langle 4, 2, 1.5 \rangle\rangle$ stating that car72 is a car object with the xyz-position $\langle 201, 75, 88 \rangle$, driving 20 m/s in the x-direction, being 4 meters long, 2 meters wide, and 1.5 meters high. A stream of such object states would be an example of an object state stream. □

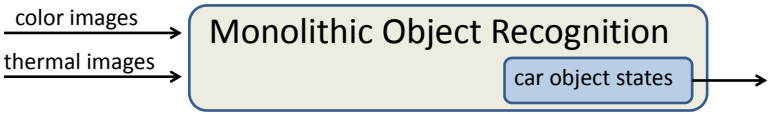


Figure 8.1: A knowledge process doing both object recognition and tracking.

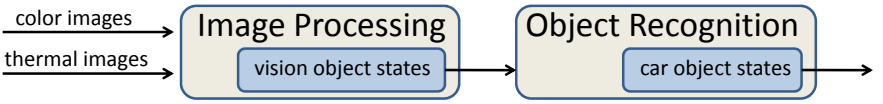


Figure 8.2: Two knowledge processes doing object recognition and tracking together.

8.5.2 Object Linkage Structure

As described in the introduction we would like to recognize and track cars and extract information such as the position, speed, and size of the tracked cars. For our UAV to acquire this information it has to process the video sequences from its onboard thermal and color cameras to recognize objects which could be cars and filter out those that are not classified as cars. Using knowledge processing middleware, this problem could be structured by introducing a knowledge process which takes a stream of images from the color camera and a stream of images from the thermal camera and produces a stream of object states where each object state represents the current state of a tracked car (Figure 8.1).

An alternative to having a single opaque knowledge process is to divide the problem into parts. Since there already exist numerous approaches to recognizing and tracking objects in video sequences the image processing is suitable for being a separate knowledge process. The concrete image processing system we use, takes as input streams of color and thermal images and outputs a stream of vision object states representing the potential cars found in the video streams (Section 8.7.1).

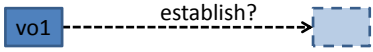
However, since image processing is not perfect and usually only does limited reasoning about the identity of the potential cars more processing is needed to increase the quality of the output. Therefore, a second knowledge process is introduced which further reasons about the type and identity of the potential cars found in the video streams. The input to this knowledge process is the stream of vision object states representing potential cars tracked by the image processing system and the output is a stream of car object states representing the current state of the tracked cars. An overview of the two knowledge processes is shown in Figure 8.2.

The task of the second knowledge process is to determine for each vision object representing a potential car if it actually is a car and in that case whether it is a known or a new car. One way of doing this reasoning is to use the temporal logic described in Chapter 7 to express an *establish condition* that holds when a new vision object represents a new car object and a *reestablish condition* that holds when it represents a known car. If one of these conditions holds, a *link* is created,

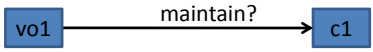
representing an association between the vision object and a new or existing car object, respectively. While a vision object is linked to a car object the attributes of the car object are computed from the attributes of the vision object. If neither condition holds, the vision object is not associated with any car object.

Since the observations about the environment are uncertain and the classification is not perfect each link is only treated as a hypothesis that the vision object and the associated car object represent the same physical object. To continually validate the hypothesis a *maintain condition* is introduced. The maintain condition is a condition which should be valid as long as the two objects are associated. If the maintain condition is violated then the link is removed to represent that the hypothesis is withdrawn and that the two objects are no longer associated. However, this does not remove the car object, only the link.

Example 8.5.2 (Link) Assume that the image processing system is currently tracking a potential car represented by the vision object vo1 and that no previous car objects have been created. Since there is a vision object but not any known car objects it is enough to evaluate the establish condition on vo1.

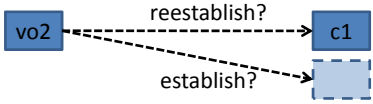


Assume that the condition is satisfied. Then a new car object c1 is created which is associated with vo1. As long as c1 is associated with vo1 its state will be computed from the state of the vision object vo1. To monitor the hypothesis that c1 is a car the maintain condition is monitored.



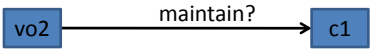
Assume the image processing system loses track of the potential car after a while. Then vo1 is removed together with the link to c1. Even though the link is removed the car object c1 still remains.

Assume further that the image processing system later recognizes a new potential car represented by the vision object vo2. Since there exists a known car, c1, the knowledge process has to evaluate whether vo2 is a new car, the known car c1, or not a car at all. This is done by evaluating the establish condition on vo2 and the reestablish condition between vo2 and c1.



Assume that after a while the establish condition is progressed to false and the reestablish condition is progressed to true. Then a new link is created from vo2 to

c1 and the attributes of c1 can be computed from the attributes of vo2. To check the new hypothesis a maintain condition between c1 and vo2 is monitored.



□

In the previous example the vision objects created by image processing were directly linked to car objects. This can easily be generalized to introduce a hierarchy of intermediary objects between vision objects and car objects. For example, vision objects can first be linked to *world objects*, where each world object represents a physical object in the world. These world objects could then be linked to *on road objects* representing physical objects that move along roads. Finally, these on road objects could be linked to car objects if they have the characteristics of cars instead of motorcycles or trucks. A set of objects linked together is called an *object linkage structure*. Not only does this support the easy integration of different object recognition and tracking approaches, it also provides an explicit representation of all the abstraction levels used to represent an object.

A specification of links from objects of type A to objects of type B consists of three conditions, the *establish*, *reestablish*, and *maintain* conditions, and a computational unit for computing B object states from A object states. A link is specified by a link declaration according to definition 8.5.3. From a link declaration a knowledge process is defined that does the object recognition and classification according to the specification.

Definition 8.5.3 (Link declaration) A *link declaration* has the form **strmgen** b = **link**(a, cu, e, r, m), where b and a are label terms, cu is a computational unit symbol, and e, r, and m are monitor formulas. □

A link specification "**strmgen** b = **link**(a, cu, e, r, m)" describes a knowledge process that subscribes to the stream generator labeled a and links the objects in the resulting stream according to the establish condition e, the reestablish condition r, and the maintain condition m. The formulas may contain the special variables *from* and *to* which refer to the object symbol of the linked from object and the linked to object respectively. To compute the object states of the linked to objects the computational unit cu is used. All the generated object states will be made available from a stream generator labeled b. The types of the objects in the streams denoted by the labels a and b are implicit but will be referred to as type A and type B. An example link process is shown in Figure 8.3 where vision objects are linked to car objects. Which car object a vision object is linked to depends on the three link conditions.

The establish condition describes when a new instance of type B should be created and linked to an instance of type A. In the traffic domain, for example, type A could correspond to world objects and type B could correspond to on road objects. When a new world object is found, the establish condition could trigger

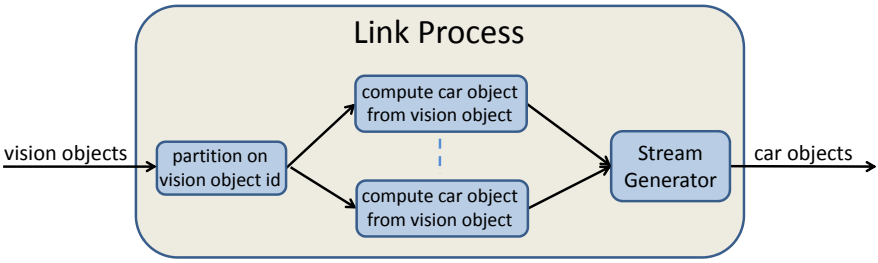


Figure 8.3: A knowledge process linking vision objects from the object state stream vision_objects to car objects provided by a stream generator labeled car_objects.

when the world object has been on the road for at least 30 seconds. If it triggers, a new on road object is created and linked to the world object. An on road object could contain more abstract and qualitative attributes such as which road segment it is on, which makes it possible to reason qualitatively about its position in the world relative to the road, other cars on the road, and building structures in the vicinity of the road. At this point, streams of data are being generated and computed for the attributes in the linked object structures at many levels of abstraction as the UAV tracks the on road objects.

The reestablish condition describes when two existing objects of the appropriate types which are not already linked should be linked. This is used when the tracking of an on road object is lost and the image processing system finds a new world object which may or may not be the same on road object as before. If the reestablish condition is satisfied then it is hypothesized that the new world object is in fact the same on road object as was previously tracked.

Since links only represent hypotheses, they are always subject to becoming invalid given additional data, so the UAV continually has to verify the validity of the links. This is done by monitoring that a maintain condition is not violated. A maintain condition could compare the behavior of the object with the normative behavior of this type of object and, if available, the predicted behavior of the previous object. In the on road object example the condition could be that the world object should remain continually on the road, maybe with occasional shorter periods off the road. If this condition is violated then the link is removed and the on road object is no longer updated since the hypothesis can not be maintained. The link is also removed if one of the objects are removed.

A link process is responsible for creating and maintaining links according to these three conditions. This is done individually for each object found in the input stream to the link process. To evaluate the conditions a link process could use the progression mechanism described in Section 7.5.2. A finite state machine describing how to link vision objects to car objects is shown in Figure 8.4 on the following page. The description is general and applies to all link processes.

For each object o found in the input stream a state machine is created which keeps track of the state for that object. When o is linked to another object o',

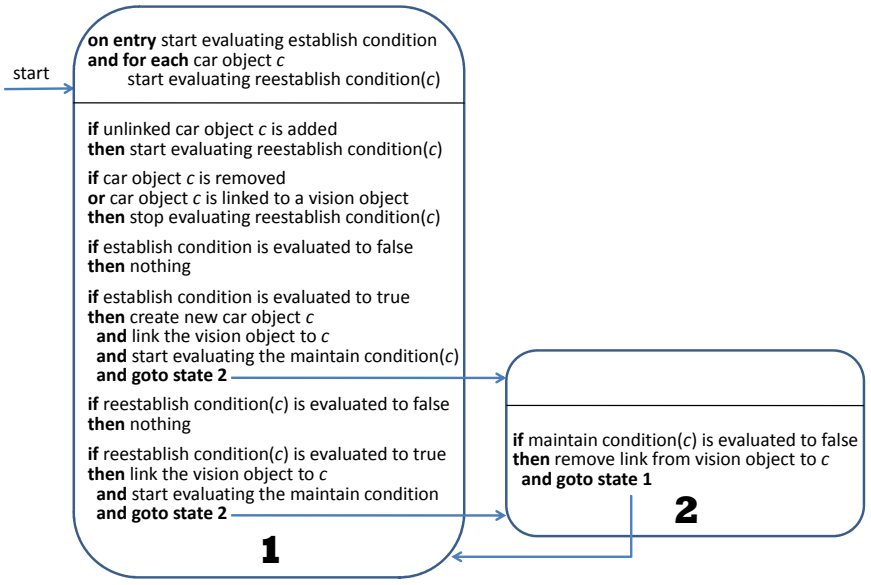


Figure 8.4: A finite state machine describing how a vision object is linked to car objects.

then an instance of the computational unit *cu* is created which takes the stream of object states for o as input and computes a stream of object states for o' as output. The output of a link process is the union of all the object state streams created by instances of *cu*. An overview of the link process which creates car objects from vision objects is shown in Figure 8.5 on the next page.

Object linkage structures are related to inheritance hierarchies in object oriented programming. A vision object is a world object which is an on road object which is a car. However, the difference is that in object oriented programming objects are created in a top-down fashion while in object linkage structures they are created bottom-up. For example, in object oriented programming if a car object is created, this object is also unavoidably an on road object and a world object. With object linkage structures, on the other hand, a vision object is first created and if certain conditions are met then a world object is created, if further conditions are met then an on road object is created, and so on.

8.6 Anchoring

We will now show how the object linkage structures introduced in the previous section can be used to anchor symbols representing objects in the world to sensor data, in such a way that the symbol is consistently associated with information about a specific physical object.

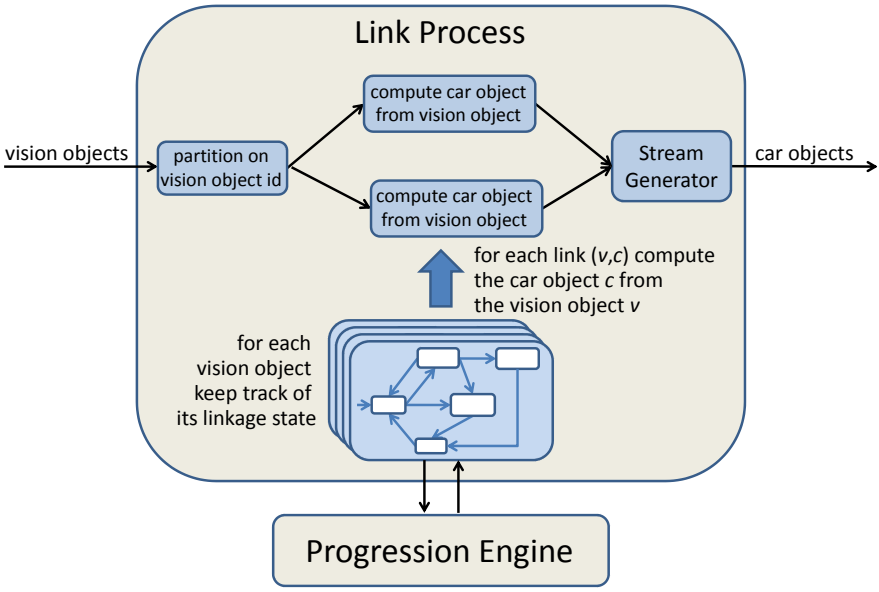


Figure 8.5: An overview of the link process which creates car objects from vision objects.

As mentioned in the introduction to this chapter, anchoring is a special case of the more general symbol grounding problem (Harnad, 1990). A definition of anchoring by Coradeschi and Saffiotti (2003) is: "We call *anchoring* the process of creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects. The *anchoring problem* is the problem of how to perform anchoring in an artificial system."

The concept of "sensor data" is very broad. Examples of sensor data are a temperature measurement from a thermometer, an image or a sequence of images from a frame grabber, a point-cloud collected by a laser range scanner or a sonar, and so on. In our approach each instance of a type of sensor data is represented by an object called a *percept*. We have already seen an example in the form of vision objects created by an image processing system. Another example could be a laser range finder that creates a point-cloud object which is updated with each scan.

In our approach, anchoring an object o is the same as creating an object linkage structure, directly or indirectly, connecting o to an object representing sensor data. As long as the object o is linked to a percept it is considered to be anchored. For example, to anchor car objects to sensor data collected by cameras in the form of vision objects, a link process connecting vision objects and car objects could be used. It is also possible to create a chain of link processes, for example first linking vision objects to on road objects and then link these on road objects to car objects. In either case, a car object would be considered anchored as long as it was connected to a vision object.

The anchoring framework suggested by Coradeschi and Saffiotti (2003) defines three anchoring functionalities: Find, Track, and Reacquire. These functionalities cover the life cycle of an anchor. The Find functionality takes a symbolic description of an object and tries to anchor it in sensor data. The Track functionality continually maintains the anchor based on the current sensor data. In case the tracking fails, the anchor is removed. The Reacquire functionality then tries to find new sensor data which corresponds to the same object that was anchored previously. The difference between Find and Reacquire is that Reacquire can use the information gathered while the anchor was being tracked. For example, the Reacquire functionality may refer to the position of an object collected while the object was anchored.

Using object linkage structures to anchor objects, these functionalities mainly correspond to evaluating the establish, reestablish, and maintain conditions associated with each link process. There is a specific link process for each type of object that can be anchored, testing conditions related to that specific object type. When a new percept p appears, there are three distinct cases.

First, the percept may be related to a new object of the given type. Therefore, DyKnow immediately begins evaluating the establish condition. If this condition eventually progresses to true, a new object o of the desired type is created and linked to the percept p. This corresponds to the Find functionality.

Second, the percept may be related to an existing but currently unlinked object of the given type. This is determined by the reestablish condition. For example, if a link process connecting vision objects to car objects has one vision object vo and two car objects c_1 and c_2 which are not linked to anything, then it would evaluate the reestablish condition on the two pairs (vo, c_1) and (vo, c_2). If one of them is evaluated to true then a link is created between the two objects, and an anchor would have been reacquired.

Third, the percept may not match the given type at all, in which case the establish condition and any reestablish conditions will eventually evaluate to false and no links will be created.

While a percept p is linked to an object o a maintain condition is being evaluated. As long as the maintain condition is not violated, p and o are linked and the current state of o is computed by a computational unit. This corresponds to the Track functionality. If the maintain condition is violated then the link between p and o is removed, but not the two objects themselves.

There are several differences between our approach of using object linkage structures and the anchoring framework proposed by Coradeschi and Saffiotti (2003). First, we do anchoring mainly bottom-up while they do it top-down. In their Find functionality a symbolic description of an object is given while our establish condition rather describes the conditions for when sensor data can be considered to refer to a particular type of object. For example, instead of describing a particular car object we use an establish constraint to describe when a vision object can be considered to be an image of a car. Second, by using a metric temporal logic to describe the conditions for anchoring time is explicitly taken into account. A third difference is that object linkage structures allow the anchoring process to be done

incrementally through several intermediary steps. This makes the anchoring problem easier to handle since sensor data, such as images, does not have to be directly connected to a car object but can be anchored and transformed in several smaller steps. Another benefit is that an object linkage structure maintains an explicit representation of all the levels of abstraction used to represent an object.

Related Work

In this section, we compare two alternative approaches to anchoring symbols which use techniques having some similarities to the DyKnow anchoring approach.

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

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

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

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

Another difference between the approaches is that Bonarini et al. compute the degree of matching for each concept in order to handle uncertain and incomplete information. Similarity measurements between objects are an essential functionality for anchoring objects to sensor data and comparing them to each other. One possibility could be to use a general theory for measuring similarity based on the

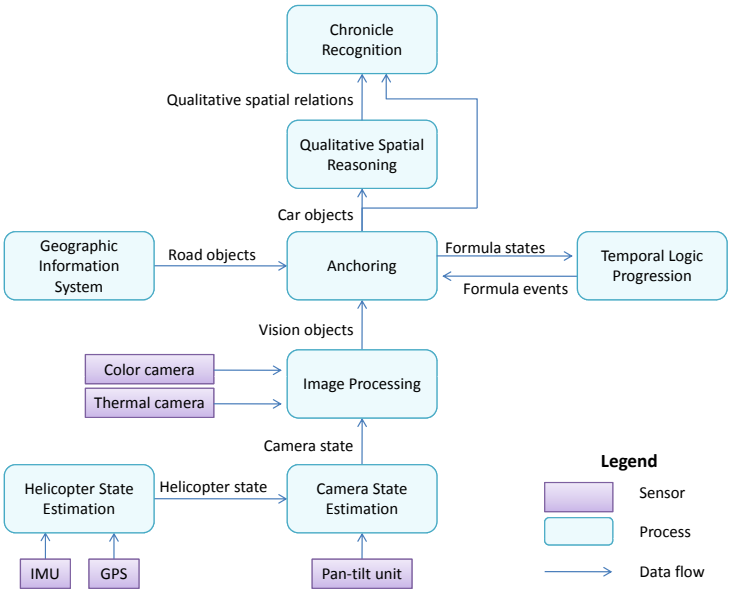


Figure 8.6: An overview of how the incremental processing for the traffic monitoring application is organized.

use of rough set techniques (Doherty and Szałas, 2004; Doherty, Łukaszewicz, and Szałas, 2003). To integrate this functionality into DyKnow is part of our ongoing activity in this area.

8.7 Implementing the Traffic Monitoring Scenario

This section provides a use case of how DyKnow can be used to implement a complete traffic monitoring application from image processing, through anchoring, to chronicle recognition of traffic situations. The inputs are images taken by the color and thermal cameras on our UAV which are fused and geolocated to a single world position. This stream of positions is then correlated with a geographical information system (GIS) in order to know where in a road system an object is located. Based on this information, high level behaviors such as turning in intersections and overtaking are recognized in real time as they develop using a chronicle recognition system. All of these functionalities are integrated using DyKnow.

An overview of all the components and the incremental processing required for the traffic surveillance task is given in Figure 8.6. All the components except the helicopter state estimation and camera state estimation are presented either in this section or in a previous chapter. Details about helicopter and camera state estimation can be found in Conte (2007).

8.7.1 Image Processing

The task of image processing in this work is to find and track cars in video sequences and calculate their world coordinates. First, an object tracker is used to find pixel coordinates of the car of interest based on color and thermal input images. Second, the geographical location of the object is calculated and expressed as world coordinates.

The object tracker can be initialized automatically or manually. The automatic mode chooses the warmest object on a road segment within the thermal camera view and within a certain distance from the UAV. The process of calculating the distance to a tracked object is explained below. The area around the initial point is checked for homogeneity in thermal and color images. The object is used to initialize the tracker if its area is consistent with the size of a car signature. This method of initialization works with satisfactory results for distances up to around 50 meters from the tracked object. If the tracker is initialized incorrectly the user can choose the object of interest manually by clicking on a frame in the color or thermal video.

The corresponding pixel position, for color and thermal images, is calculated based on the parameters of the cameras, the UAV's position and attitude, and the model of the ground elevation. After initialization tracking of an object is performed independently in the color and thermal video streams. Tracking in the thermal image is achieved by finding the extreme value (warmest or coldest spot) within a small (5 percent of the image size) window around the previous result.

Object tracking in color video sequences is also performed within such a small window and is done by finding the center of mass of a color blob in the hue, saturation, and intensity (HSI) color space. The thresholding parameters are updated to compensate for illumination changes. Tracking in both images is performed at full frame rate (25 Hz) which allows for compensating for moderate illumination changes and moderate speeds of relative motion between the UAV and the tracked object. The problem of automatic reinitialization in case of loss of tracking, as well as more sophisticated interplay between both trackers, is not addressed in this work. The result from the thermal image tracking is preferred if the trackers do not agree on the tracking solution.

In order to find the distance to the tracked object as well as corresponding regions in both images, the cameras have been calibrated to find their intrinsic and extrinsic parameters. The color camera has been calibrated using the Matlab Camera Calibration Toolkit (Bouguet, 2000). The thermal camera has been calibrated using a custom calibration pattern and a different calibration method (Wengert et al., 2006) because it was infeasible to obtain sharp images of the standard chessboard calibration pattern. The extrinsic parameters of the cameras were found by minimizing the error between calculated corresponding pixel positions for several video sequences.

Finding pixel correspondences between the two cameras can not be achieved by feature matching commonly used in stereovision algorithms since objects generally appear differently in color and infrared images. Because of this fact, the distance to an object whose projection lies in a given pixel must be determined.

Figure 8.7: A. Two frames from a video sequence with the UAV hovering close to a road segment observing two cars performing an overtaking maneuver. B. Three frames from a video sequence with the UAV following a driving car passing road crossings. The top row contains color images and the bottom row contains corresponding thermal images.

Given the camera parameters, helicopter pose, and the ground elevation model the distance to an object can be calculated. It is the distance from the camera center to the intersection between the ground model and the ray going through the pixel belonging to the object of interest. For the environment in which the flight tests were performed the error introduced by a flat world assumption (i.e. ground elevation model simplified to a plane) is negligible. Finally, calculating pixel correspondences between the two cameras can be achieved by performing pixel geolocalisation using intrinsic and extrinsic parameters of one of the cameras followed by applying an inverse procedure (i.e. projection of geographical location) using the other camera parameters.

Using the described object tracker, several data series of world coordinates of tracked cars were generated. Two kinds of video sequences were used as data sources. In the first kind (Figure 8.7A) the UAV is stationary at an altitude between 50 and 60 meters and observes two cars as they drive on a nearby road. In the other kind (Figure 8.7B) both the car and the UAV are moving. The ground car drives several hundred meters on the road system passing through two crossings and the UAV follows the car at an altitude between 25 and 50 meters. For sequences containing two cars, the tracker was executed twice to track each car independently.

A precise measure of the error of the computed world location of the tracked object is not known because the true location of the cars was not registered during the flight tests. The accuracy of the computation is influenced by several factors, such as the error in the UAV position and the springs in the camera platform suspension, but the tracker in general delivers world coordinates with enough accuracy to determine which side of the road a car is driving on. Thus the maximum error can be estimated to be below 5 meters for distances to the object of around 80 meters. For example results of car tracking see Figure 8.9 and Figure 8.13.

8.7.2 Anchoring

The anchoring component of the traffic monitoring application takes the stream of potential cars from the image processing system and tries to determine which of these objects actually are cars.

In the implemented approach, the image processing system produces *vision objects* representing those entities (called *blobs*) found in an image that have visual and thermal properties similar to a car. A vision object state contains an estimation of the size of the blob (length and width), its position in absolute world coordinates (pos), and a predicate stating whether this position is on the road system according to the GIS or not (on_roadsystem). When a new vision object has been found it is given a unique object identifier and it is tracked for as long as possible by the image processing system. Each time the tracker finds the same object in an image, a new vision object state with the same object identifier is pushed on a stream called vision_objects.

To anchor these vision objects to car objects a link process is defined by the following link specification:

strmgen cars = **link**(vision_objects,
ComputeCarObject,
$\Diamond\,\Box_{[0,1000]}$ on_roadsystem(*from*),
$\perp$,
$\Box\,\Diamond_{[0,30000]}\,\Box_{[0,5000]}$ on_roadsystem(*from*)
)

The specified link process takes the output stream from the image processing, vision_objects, and provides a stream generator called cars containing the car objects computed from these vision objects. The computational unit which transforms a vision object into a car object is called ComputeCarObject. The establish constraint, $\Diamond\,\Box_{[0,1000]}$ on_roadsystem(*from*), states that a vision object must be observed on the road system for at least 1000 milliseconds before assuming that it is a car. In the current implementation we do not reacquire anchors so the reestablish condition is $\perp$. The maintain condition, $\Box\,\Diamond_{[0,30000]}\,\Box_{[0,5000]}$ on_roadsystem(*from*), states that it is always the case that within 30 seconds an interval of at least 5 seconds starts where the vision object is observed on the road all the time.

8.7.3 Integrating Chronicle Recognition

In order to use chronicle recognition to recognize event occurrences the event must be expressed in the chronicle formalism and a suitable stream of primitive events must be generated. Since a primitive event is defined as a change in the value of an attribute, it is enough to subscribe to the appropriate fluent stream and create a primitive event each time the value changes. The only requirement on the fluent stream is that the samples arrive ordered by valid time. The reason is that all the information for a specific time-point has to be available before the chronicle can be updated with this new information. This means that whenever a new sample arrives with the valid time t the chronicle is propagated up to the time-point $t - 1$ and then

the new information is added. If a sample arrives out of order it will be ignored. The integration is done in two steps, integration of chronicles and integration of events.

The first step is when a chronicle is registered for recognition. To integrate a new chronicle DyKnow goes through each of the attributes in the chronicle and subscribes to the corresponding fluent stream. Each attribute is considered a label for a fluent stream generator producing discrete values. To make sure that the chronicle recognition engine gets all the intended changes in the fluent stream ordered by valid time a policy with a monotone order constraint is used when subscribing.

The second step is when a sample arrives to the chronicle recognition engine. To integrate a sample it must be transformed into an event, i.e. a change in an attribute. To do this, the recognition engine keeps track of the last value for each of the attributes and creates an event when the attribute changes values. The first value is a special case where the value changes from an unknown value to the first value. Since it is assumed that the events arrive in order the recognition engine updates its internal clock to the time-point before the valid time of each new sample. In this manner the chronicle engine can prune all partial chronicles which can no longer be recognized.

8.7.4 Intersection Monitoring

The first part of the traffic monitoring application is to monitor activities in an intersection. In this case the UAV stays close to an intersection and monitors the cars going through. Each car should be tracked and it should be recorded how it travelled in the intersection to create a stream of observations such as ”car c came from road a to crossing x and turned left onto road b”. The cars are tracked by the vision system on the UAV and the information about the road system comes from a GIS. This section describes how this information is fused and how it is used to recognize the behavior of the cars in real-time as the situation develops.

The road system is represented in the GIS as a number of *areas* which cover the road system. Each area is classified as either being a crossing or a road (in Figure 8.8 and Figure 8.9 the green areas are the crossings and the yellow are roads). There are different areas representing the different lanes of a road. To represent a road connecting two crossings an abstraction called a *link* is introduced. All road areas between two crossings are part of the link. The separation of areas and links is made in order to be able to reason both about the geometry and other low level properties of the roads and higher level road network properties. The geometry is for example needed in order to simulate cars driving on roads and to find the road segment given a position. The network structure is for example needed when reasoning about possible routes to a destination.

To represent the possible turns that can be made from a link a class of objects called Link is used. All Link objects have the four attributes, left, right, straight, and uturn. Since it is possible to turn for example left in many different ways the value of the attribute is a sets of triples $\langle \mathbf{link1}, \mathbf{crossing}, \mathbf{link2} \rangle$ where each triple

represents that a car going from **link1** through **crossing** to **link2** has made a left turn. The link object states are made available by a stream generator called links.

These relations are somewhat clumsily represented by attributes in a link object. Each relation is a sequence of sequences of strings, which represents a set of tuples where each tuple contains the identifiers of the last two objects in the triple described above. The first object in the triple is always the link identifier. For example, a link l28 with the attribute left with the value ⟨⟨ x1, l12 ⟩, ⟨ x5, l84 ⟩⟩ represents the triples ⟨⟨ l28, x1, l12 ⟩, ⟨ l28, x5, l84 ⟩⟩.

The information about the cars observed by the UAV will be provided by the link process doing the anchoring. Each car object produced by this link process has the attributes pos, link, crossing, and drive_along_road. The attribute pos is the position from the vision object linked to the car object. The attribute link is the identifier of the link the car is on according to the GIS. If the position is not on a link then the value is `no_link`. The attribute crossing is similar to the link attribute but has a value if the area is a crossing, otherwise the value is `no_crossing`. This means that the car is not on the road system if the link attribute is `no_link` and the crossing attribute is `no_crossing` at the same time. The drive_along_road attribute will be explained in the next section.

The information about the world is thus provided as two object state streams, one containing information about links and one about cars. In order to detect the intersection behavior of the cars these streams must be further analyzed. In this application chronicle recognition is used to describe and recognize behaviors. The chronicle for detecting left turns is shown below.

chronicle turn_left_in_crossing[*?c*,*?l1*,*?x*,*?l2*]
{
  occurs(1,1,cars.link[*?c*]:(*?l1*,`no_link`),(t2,t3))
  occurs(1,1,cars.crossing[*?c*]:(`no_crossing`,*?x*),(t2,t3))

  event(cars.crossing[*?c*]:(*?x*,`no_crossing`),t4)
  event(cars.link[*?c*]:(`no_link`,*?l2*),t5)

  noevent(cars.link[*?c*]:(*?l1*, `no_link`), (t3+1, t5-1))
  noevent(cars.crossing[*?c*]:(`no_crossing`, *?x*), (t3+1, t5-1))
  noevent(cars.crossing[*?c*]:(*?x*, `no_crossing`), (t2, t4-1))
  noevent(cars.link[*?c*]:(`no_link`, *?l2*), (t2, t4-1))

  event(links.left[*?l1*,*?x*,*?l2*]:(*?*,true),t1)
  noevent(links.left[*?l1*, *?x*, *?l2*]:(*?*, true), (t1+1, t5-1))
  noevent(links.left[*?l1*, *?x*, *?l2*]:(true, false), (t1+1, t5-1))

  t1 < t2
  t3-t2 **in** [-1000, 1000]
  t4-t3 **in** [0, 10000]
  t5-t4 **in** [-1000, 1000]
}

The chronicle states that a car makes a left turn if it is on link *?l1*, enters crossing *?x*, leaves on link *?l2*, and the triple ⟨ *?l1*, *?x*, *?l2* ⟩ constitutes a left turn according to the GIS. The label cars.link[car1] refers to the link attribute of a Car object with the object identifier car1 found in the cars stream. The temporal constraints at the end assert that the car should be observed to be in the crossing within 1 second before or after it has been observed not to be on any link and that the turn should not take more than 10 seconds to make. The chronicle also contains a number of **noevent** statements to make sure that no changes occur between the entering of the crossing and the leaving of the crossing.

Since the link attribute is quite coarse it is possible to manage the uncertainty in the position of the car which causes it to be on the correct link but not in the correct lane. It is also possible to define a chronicle to detect turns which are made from the correct lane, but this will often fail due to noise. For example, see Figure 8.9, where the trajectory of a tracked car is shown as it drives through an intersection.

The chronicle will fail if no observation is made of the car in the crossing, which can happen when the speed of the car is too high or the time between observations is too long. To predict that a car actually turned in the crossing even though it was only observed on a link before the crossing and on a link after the crossing the following chronicle is used:

```
chronicle turn_left_in_crossing_predicted[?c,?l1,?x,?l2]
{
    event(cars.link[?c]:(?, ?l1), t2)
    event(cars.link[?c]:(?l1, ?l2), t3)
    noevent(cars.link[?c]:(?l1, ?), (t2+1, t3-1))

    event(links.left[?l1, ?x, ?l2]:(?, true), t1)
    noevent(links.left[?l1, ?x, ?l2]:
            (?, true), (t1+1, t3-1))
    noevent(links.left[?l1, ?x, ?l2]:
            (true, false), (t1+1, t3-1))

    t1 < t2
    t2 < t3
}
```

This chronicle is much simpler than the one before since it only checks that the car passed from one link to another and that according to the GIS this transition indicates that the car must have passed a crossing and actually made a left turn. This is an example of where qualitative information about the road system can be used to deduce that a car must have passed through the crossing even though this was never observed.

There is still one issue which is illustrated by Figure 8.8. Due to noise in the position of the tracked car, it looks like the car enters the crossing, leaves the crossing, and then comes back. These types of oscillating attributes are very common in the transition between two values of an attribute.

Figure 8.8: An example where noise makes it look like a car enters a crossing twice. Each red dot is an observed car position.

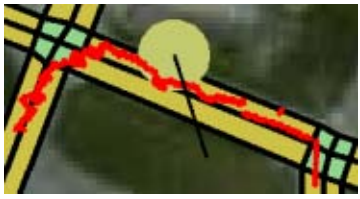


Figure 8.9: An example intersection situation recorded during a test flight.

A solution is to introduce a filter which only changes the value of an attribute if it has been stable for a fixed amount of time, in our case 500 milliseconds. Since a car is not expected to change links very often (a link is usually several hundred meters long even though shorter links may exist in urban areas) it is reasonable to say that the value of the link attribute must be stable for half a second. One possible issue is when a car is not in a crossing for more than 500 milliseconds, but this case will be detected by the predicted turn chronicle so the turn will be detected in any case.

Several other solutions could be thought of, for example to use fuzzy logic to represent the uncertainty in the value of an attribute. However, since we are interested in building an integrated traffic monitoring application each particular solution is not that important. What is important is to get a working application.

Using the setup and the chronicles described above it is possible to detect all the turns made by one or more cars driving in an urban area using either simulated cars or cars tracked by our UAV platform during test flights. One particular trajectory from a test flight where two left turns are recognized is shown in Figure 8.9.

8.7.5 Road Segment Monitoring

The second monitoring task involves the UAV observing a road segment and collecting information about the behavior of the cars passing by. Here, the focus is on recognizing overtakes. However, this is just an example, other behaviors could be detected in the same way. To recognize overtakes a stream of qualitative spatial relations between pairs of cars, such as behind and beside, is computed and used as input to the chronicle recognition system.

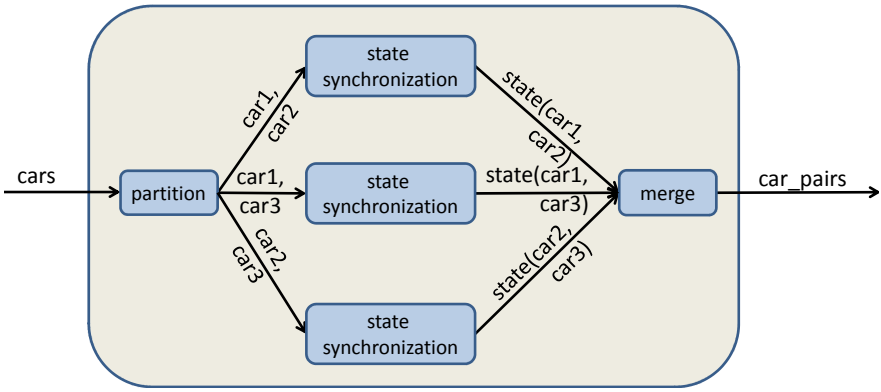


Figure 8.10: The synchronization of car pairs.

This might sound like a very simple task, but does in fact require a number of steps. First, the set of cars that are actually being tracked must be extracted from the stream of car observations and then the set of pairs of active cars can be computed from those. Second, for each pair of car identifiers a stream of synchronized pairs of car object states has to be created. Since they are synchronized both car states in the pair are valid at the same time-point, which is required to compute the relation between two cars. Third, from this stream of car pairs the qualitative spatial relations must be computed. Finally, this stream of car relations can be used to detect overtakes and other driving patterns using the chronicle recognition engine. All these functions are implemented as computational units.

To extract the active cars a computational unit is created which keeps track of all car identifiers which have been updated the last minute. This means that if no observation of a car has been made in more than 60 seconds it will be removed from the set of active cars. For example, assuming the stream of car object states looks like $\langle\langle t_1, \langle \mathsf{car1}, \ldots\rangle\rangle, \ldots, \langle t_2, \langle \mathsf{car2}, \ldots\rangle\rangle, \ldots, \langle t_3, \langle \mathsf{car3}, \ldots\rangle\rangle, \ldots\rangle$ and $t_3 - t_1 \leq 60$ seconds, then the stream of sets of active cars would be $\langle\langle t_1, \{\mathsf{car1}\}\rangle, \langle t_2, \{\mathsf{car1}, \mathsf{car2}\}\rangle, \langle t_3, \{\mathsf{car1}, \mathsf{car2}, \mathsf{car3}\}\rangle\rangle$.

Since the qualitative relations that are computed are symmetric and irreflexive the computational unit that extracts pairs of car identifiers only computes one pair for each combination of distinct car identifiers. To continue the example, the stream of sets of pairs would be $\langle\langle t_1, \{\}\rangle, \langle t_2, \{\langle \mathsf{car1}, \mathsf{car2}\rangle\}\rangle, \langle t_3, \{\langle \mathsf{car1}, \mathsf{car2}\rangle, \langle \mathsf{car1}, \mathsf{car3}\rangle, \langle \mathsf{car2}, \mathsf{car3}\rangle\}\rangle\rangle$. The stream of sets of pairs is called CarPairs and is updated when a car is added or removed from the set of active car identifiers, called Cars. This stream of car identifier pairs is then used as input to a state extraction computational unit which for each pair synchronizes the corresponding streams of car object states as shown in Figure 8.10.

Finally the car pair object states are used as input in the car relation computational unit which computes the qualitative spatial relations between the two cars. Since the spatial relation between the cars should be relative to the road network

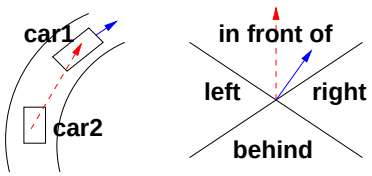


Figure 8.11: The qualitative spatial relations used.

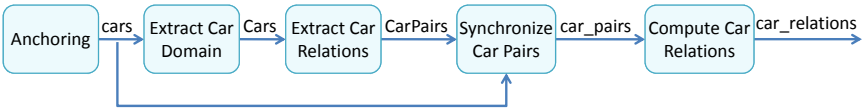


Figure 8.12: The DyKnow setup used in the overtake monitoring application.

we can not compare the driving directions of the two cars directly. Instead we compare the forward direction of car1 with the direction from car2 to car1, as shown in Figure 8.11, to determine the spatial relation between car1 and car2. In the example car1 is in front of car2. The forward direction of the car is assumed to be either along the current road segment or against it. To compute which, the current direction of the car as estimated by the derivative of the position of the car, is compared to the forward direction of the road segment.

The chronicle that is used to detect overtakes is shown below. It detects that car1 is first behind car2 and then in front of car2. A requirement that they are beside each other could be added to strengthen the definition.

chronicle overtake[*?car1*, *?car2*]
{
  event(car_relations.behind[*?car1*, *?car2*]:(*?*, true), t1)
  event(car_relations.behind[*?car1*, *?car2*]:(true, false), t2)
  event(car_relations.in_front[*?car1*, *?car2*]:(false, true), t3)
  noevent(car_relations.behind[*?car1*, *?car2*]:(true, false), (t1, t2 − 1))
  noevent(car_relations.behind[*?car1*, *?car2*]:(false, true), (t2 + 1, t3 − 1))
  noevent(car_relations.in_front[*?car1*, *?car2*]:(false, true), (t1, t3 − 1))
  t1 < t3
  t1 < t2
}

The complete setup is shown in Figure 8.12. The recognition has been tested on both simulated cars driving in a road system and on real data captured during flight tests. One example of the latter is shown in Figure 8.13.

8.7.6 Experimental Results

The traffic monitoring application has been tested both in simulation and on images collected during flight tests, an example of which was shown in Figure 8.13. The

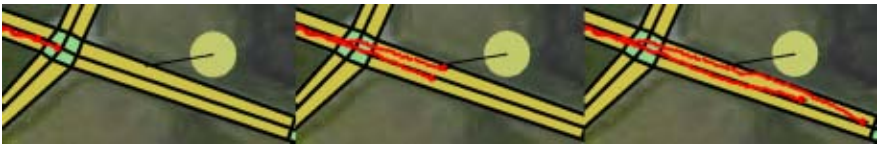


Figure 8.13: An example overtake situation recorded during a test flight.

Speed	**Sample period**	**0 m error**	**2 m error**	**3 m error**	**4 m error**	**5 m error**	**7 m error**
15 m/s	200 ms	100%	70%	60%	50%	40%	20%
20 m/s	200 ms	100%	60%	70%	50%	50%	20%
25 m/s	200 ms	100%	100%	50%	40%	10%	0%
15 m/s	100 ms	100%	60%	90%	60%	90%	0%
20 m/s	100 ms	100%	90%	90%	90%	80%	20%
25 m/s	100 ms	100%	90%	80%	70%	80%	0%

Table 8.2: The results when varying the car speed and the sample period.

only difference between the two cases is who creates the world objects.

The robustness to noise in the position estimation was tested in simulation by adding random errors to the true position of the cars. The error has a uniform distribution with a known maximum value e and is added independently to the x and y coordinates, i.e. the observed position is within an $e \times e$ meter square centered on the true position. Two variables were varied, the speed of the car and the sample period of the position. For each combination 10 simulations were run where a car overtook another. If the overtake was recognized the run was considered successful. The results are shown in Table 8.2.

The conclusions from these experiments are that the speed of the car is not significant but the sample period is. The more noise in the position the more samples are needed in order to detect the overtake. Since the estimated error from the image processing is at most 4-5 meters the system should reliably detect overtakes when using a 100 millisecond sample period.

8.7.7 Related Work

There is a great amount of related work which is relevant for each of the components, but in the spirit of the thesis the focus will be on integrated systems. There are a number of systems for monitoring traffic by interpreting video sequences, for example Ghanem et al. (2004), Nagel, Gerber, and Schreiber (2002), Medioni et al. (2001), Fernyhough, Cohn, and Hogg (1998), Chaudron et al. (1997), and Huang et al. (1994). Of these, almost all operate on sequences collected by static surveillance cameras. The exception is Medioni et al. (2001) which analyses sequences collected by a Predator UAV. Of these none combine the input from both color and thermal images.

A second major difference is how the scenarios are described and recognized. The approaches used include fuzzy metric-temporal logic (Nagel, Gerber, and Schreiber, 2002), state transition networks (Fernyhough, Cohn, and Hogg, 1998), belief networks (Huang et al., 1994), and Petri-nets (Chaudron et al., 1997; Ghanem et al., 2004), none of which have the same expressivity when it comes to temporal constraints as the chronicle recognition approach we use.

A third difference is the ad-hoc nature of how the components of the system and the data flow are connected. In our solution the basis is a declarative description of the properties of the different data streams which is then implemented by DyKnow. This makes it very easy to change the application to e.g. add new features or to change the parameters. The declarative specification could also be used to reason about the system itself and even modify it at run-time.

8.8 Summary

This chapter has provided a detailed account of how to recognize objects and chronicles in an implemented traffic monitoring application. The system implemented takes as input sequences of color and thermal images used to construct and maintain qualitative object structures and recognize the traffic behavior of the tracked cars in real time. The system is tested both in simulation and on data collected during test flights.

We believe that this type of system where streams of data are generated at many levels of abstraction using both top-down and bottom-up reasoning handles many of the issues related to closing the sense reasoning gap. A reason is that the information derived at each level is available for inspection and use. This means that the subsystems have access to the appropriate abstraction while it is being continually updated with new information and used to derived even more abstract structures. High level information, such as the type of car, can then be used to constrain and refine the processing of lower level information. The result is a very powerful and flexible system capable of achieving and maintaining high level situation awareness.

Chapter 9

DyKnow Federations

9.1 Introduction

In many robotic applications, it is not enough to only have one agent executing a mission. In a UAV application for example, there is sometimes no single UAV that has the capability or the information to perform all the required tasks. In many cases, it is also more efficient to use multiple UAVs to complete a mission. Therefore it would be beneficial for groups of UAVs to accomplish complex missions in a cooperative manner. Since the UAVs have their own limited fields of view and spheres of influence they must share and merge information among themselves to cooperatively complete missions. The information could include plans, observations, and partial world models.

Conventional approaches to merging and fusing information have focused on collecting information from distributed sources and processing them at a central location. Our goal is to allow each platform to be autonomous and to do as much processing as possible locally even when they have agreed that they will need to cooperate to solve a particular task. This will make the processing more decentralized and remove the dependence on a central node with global information. This goal can be divided into four separate subproblems:

1. How to find and share information among nodes,
2. how to merge information from multiple sources,
3. how to jointly decide that cooperation is necessary, and
4. how to divide the information processing among the multiple nodes given a joint goal.

In Chapter 10 we will show how DyKnow can be used to implement most of the JDL Data Fusion Model, which is the de facto standard functional fusion model (Llinas et al., 2004; Steinberg and Bowman, 2001; White, 1988). This shows that

DyKnow has the necessary functionality to support fusing and merging information. Since the main topic of this thesis is knowledge processing the focus of this chapter is to show how DyKnow can be extended to support finding and sharing information among multiple UAVs. The problems of how to reach a common agreement about joint goals and how to cooperate to achieve these goals are interesting issues but outside the scope of this thesis.

The rest of the chapter is structured as follows. Section 9.2 describes two motivating scenarios and some specific use cases where sharing and merging information is required. Section 9.3 describes the distributing infrastructure where DyKnow instances from participating platforms are connected in a DyKnow federation. The federation is created and controlled using a FIPA (Foundation for Intelligent Physical Agents) compliant multi-agent framework. Section 9.4 describes an implementation of a multi-UAV proximity monitoring functionality to show how the DyKnow federation can be used. Section 9.5 concludes the chapter with a summary.

9.2 Motivating Scenarios

Before describing the details of the DyKnow federation framework, two motivating scenarios are presented. The first scenario is a proximity monitoring scenario and the second is a multiple platform traffic monitoring scenario.

9.2.1 Proximity Monitoring

For a UAV to operate safely there are many conditions that need to be monitored. For example, it must have enough fuel and battery power to complete its mission and the airspace around it must be free from obstacles. Since our UAV is currently not equipped with a physical proximity sensor it is necessary to have a virtual proximity sensor which monitors the distance between a UAV and all other UAVs in the vicinity. This is a good example where UAVs need to share information in order to monitor a safety constraint.

To start the proximity monitoring, a UAV called uav1, is given the goal to monitor that no pair of UAVs are getting too close to each other. To achieve this goal one possibility is to collect position information from all UAVs to a central monitoring agent. Another approach is to delegate to each UAV the goal of monitoring its own proximity to other UAVs. In this case, each UAV will monitor that it does not get too close to any other UAV. For this the UAV needs periodic information about the current position of all other UAVs in the area. To collect this information uav1 will delegate the goal of broadcasting the current position with a certain period to each of the other UAVs. To achieve the goal of periodically broadcasting its current position a UAV sets up a DyKnow federation with the other UAVs where the position is pushed to them with the desired sample period. The position of a UAV is estimated by merging local information from sensors such as GPS (Global Positioning System) and IMU (Inertial Measurement Unit) using fusion techniques such as Kalman filters (Kalman, 1960).

Using the DyKnow federation framework each UAV will periodically get the current position of all the other UAVs. This information is then used to construct a partial temporal logical model of the environment. This incrementally constructed model is used by a temporal logic progression engine to evaluate a temporal logical formula which captures the safety constraint, as described in Chapter 7. If two UAVs come too close to each other a proximity violation alarm is raised and the UAVs have to react accordingly to avoid a potential collision.

9.2.2 Traffic Monitoring with Multiple UAVs

Assume that two or more UAVs are given the task of monitoring an urban area for traffic violations. Each UAV is equipped with the appropriate sensors and reasoning mechanisms for detecting traffic violations. This means that each UAV could monitor and detect traffic violations by itself, if it sees the whole situation. This could for example be done as described in Chapter 8.

To increase the size of the monitored area or to monitor several different potential traffic violations at the same time, several UAVs can be used. Even a simple approach to cooperation like dividing the area between the UAVs introduces issues related to sharing and merging information. For example, different characteristics of the UAVs could be used to decide how to divide the area, such as their speed, flying altitude, sensors, and available fuel. If one UAV is responsible for dividing the area it will need to collect this information from all platforms.

Another issue is the possibility of a traffic violation beginning in one sub-area and ending in another. In this situation, neither of the UAVs will see the whole event. To handle this situation the UAVs need to cooperate and share information in such a way that they can detect the traffic violation together. One approach is to let the UAV that detected the beginning of the potential violation request the appropriate information from the UAV responsible for the area where the vehicles are headed. What is appropriate will depend on how traffic violations are detected, one approach could be to share the position information about the tracked vehicles. This information would have to be seen as a stream since it is not a single piece of information but rather an evolving description of the development of a complex situation. Merging such a stream with local information would allow the first UAV to detect the traffic violation even if it takes place in two different areas.

This traffic monitoring scenario is an instance of a class of scenarios where multiple platforms must cooperate to complete complex missions. To succeed they need to collect, share, and merge information. A solution which handles the issues introduced in this scenario will also provide a solution for many other interesting scenarios. For example, instead of having homogeneous platforms covering different parts of an area there could be heterogeneous platforms with complementing sensors each providing different types of information. Another example is to increase the accuracy in the monitoring by having several homogeneous or heterogeneous platforms covering the same area. It is also possible to replace traffic monitoring with scanning an area for injured people to do a rescue mission or to look for troops and military equipment to do a military surveillance mission.

9.2.3 Design Requirements

When designing a framework for sharing information among multiple nodes there are several important issues that need to be considered.

First, how to refer to a piece of information when communicating with other nodes, i.e. how to handle naming issues. This is the problem of how to agree on a common ontology among a group of nodes. The ontology is required for a node to be able to refer to a particular piece of information when talking to other nodes.

Second, how to discover information among a group of nodes. When a node needs a specific piece of information that it does not have, then it needs to find another node which is able to deliver it. Such a mechanism should be able both to find a node who either has or can produce a particular piece of information and to announce to interested nodes when a particular piece of information is available.

Third, how to negotiate with other nodes to make them generate desired information. In the simplest form this mechanism would request the production of a piece of information from a node. In the general case a node could refuse to perform the request due to limited resources or conflicting commitments. There might also be several nodes that could produce the same information but with different quality and costs. In this case a node would have to reason about the different options and negotiate with the nodes to find a node who is willing to produce the information with good enough quality while limiting the cost.

Fourth, how to deliver information from one node to one or more other nodes interested in the information. The mechanism should allow for a robust and efficient transfer of information between nodes while taking the properties of the communication medium into account, such as the risk of losing or corrupting messages, low bandwidth, or a single shared channel.

To make these requirements more explicit three different use cases are presented. Together they cover most of the functionality required for the proximity monitoring and multi-platform traffic monitoring scenarios. The DyKnow federation framework provides an integrated framework with basic support for implementing these use cases.

Explicit Ask and Tell to Divide the Monitoring Area

To divide an area to be monitored among a group of UAVs they need to negotiate. One approach would be to appoint one of them the leader. This leader then has to find out which UAVs are available and collect information about them. The information could for example be available sensors and the maximum speed and flying altitude. Using this information the leader can partition the area among the UAVs and inform them about their responsibilities.

This use case gives an example where a node needs to find which other nodes are available, ask for specific pieces of information from each of the nodes, compute the result, and then inform the other UAVs about the result.

Continuous Information Streaming to Detect Traffic Violations

When monitoring a traffic violation occurring in two adjacent areas covered by different UAVs there will be an interval where none of the UAVs has a complete picture of the situation. This means that they have to cooperate with each other in order to observe the whole development.

A concrete use case is when a UAV has detected the beginning of a potential traffic violation involving two cars and one of the cars leaves the view field of UAV A, entering the view field of UAV B. Now, UAV A has to continuously get relevant updates from UAV B about the car it can no longer see.

The information provided by UAV B could be on many different abstraction levels. A high abstraction level in this case could be to send a stream of car states with the best current estimation of the position of the car. UAV A can then merge the information received from UAV B with the information gathered by its own sensors in order to monitor the potential traffic violation. It is important to notice that this is an ongoing activity where each new car state should be transmitted to UAV A to be merged with each car state it produces locally.

Another example is to merge information on a lower level. Instead of letting UAV B produce car states it could share more primitive information. The lowest possible level would be to send the raw sensor data, such as images. This will in most cases not be appropriate since communication bandwidth is limited.

Note that in this example, both UAVs were assumed to have identical abilities. In the general case, however, heterogeneous processing capabilities may affect the abstraction level of data being shared.

To find an appropriate abstraction level for the communication many factors must be taken into account. The most important ones are the processing capability of the involved platforms, the available bandwidth, and the current commitments of the involved platforms. In general we believe that the higher the abstraction level the less information needs to be shared and the easier it is to merge it with existing information.

Merging Information to Get a Global Picture

A slightly different use case is if an operator would like to have all the information about all the tracked vehicles in an area. In this use case a number of UAVs are looking for and tracking vehicles. Each UAV creates its own local identifiers for the vehicles it has found. When a UAV detects a vehicle it has not seen before, it should be reported to the operator. As long as the UAV is tracking the car the operator should receive continuous updates about the estimated car state.

One question is now whether the vehicle found by the UAV is the same as one of the vehicles the operator already has information about. To merge the information from all the available nodes it is therefore necessary to reason about the identities of the tracked vehicles. Which are the same? If two identifiers refer to the same vehicle then the information related to these identifiers should be merged.

This use case can be made more interesting by adding and removing nodes. Each time a new node is added then any vehicle which is tracked by that node

should be reported back to the operator. If a node is removed then the operator should be notified that the information from that UAV is no longer available.

9.3 Sharing Information using DyKnow

From the point of view of DyKnow, multiple physical platforms could be viewed as sharing a single instance of DyKnow, since DyKnow is designed for a distributed environment and does not differentiate between streams based on where they are hosted or generated. However, much of the information processed by DyKnow will be local to a single platform. It would therefore incur an overhead to communicate with a single central DyKnow instance. With multiple DyKnow instances, one for each platform, this overhead is avoided.

Each DyKnow instance requires a single specification with unique names for sources, computational units, stream generators, and streams. In a distributed system without global control it is non-trivial but doable to support unique names, for example, by relying on a common naming schema or a common service for creating new names. By having a DyKnow instance for each platform the coupling between the nodes is looser and it becomes easier to add new nodes and to implement the different nodes independently.

Another benefit with many DyKnow instances is that only relevant information needs to be shared among the instances. This is appropriate since the internal structures and representations used by one node should not necessarily be public to all other nodes. Most of each local specification will be irrelevant to other nodes and some should even be kept secret. By only sharing the relevant information, the communication overhead is further reduced and the robustness is increased since the system does not require reliable and stable communication all the time.

We will therefore extend DyKnow to allow different DyKnow instances to be developed and used independently and then connected in a federation on demand. When nodes are connected, parts of their local DyKnow instances are shared among them.

9.3.1 DyKnow Federation Overview

To fulfill the requirements introduced in Section 9.2.3 we propose to connect nodes having local DyKnow instances in a DyKnow federation, similar to the concept of federated databases (Heimbigner and Mcleod, 1985; Sheth and Larson, 1990). The federation is used to find other DyKnow instances which can provide a particular piece of information and to ask queries about information available at other nodes. To support efficient continuous streaming of information between nodes we propose to create direct communication channels on-demand between pairs of nodes. These channels are set up through the federation framework but are then under the control of the participating nodes. From the perspective of a local DyKnow instance information from remote nodes is treated as if it were local. A high level overview of a DyKnow federation is shown in Figure 9.1.

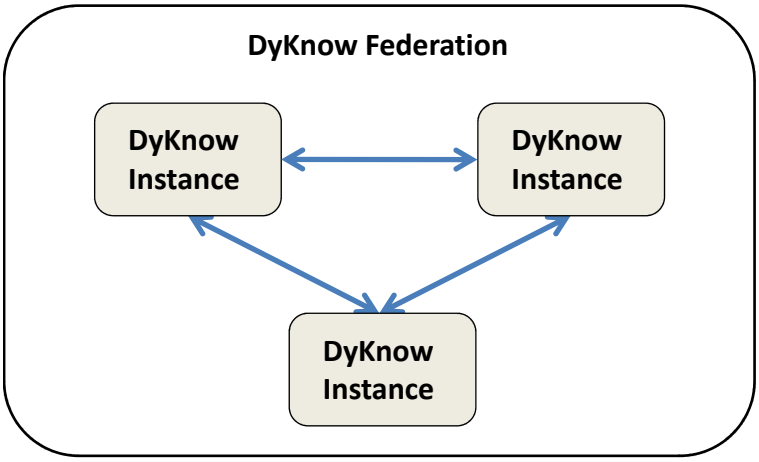


Figure 9.1: A high level overview of a DyKnow federation.

The DyKnow federation framework uses an existing multi-agent framework (Doherty and Meyer, 2007), where each DyKnow instance becomes a service. A DyKnow federation is managed through speech act-based interactions between these services.

9.3.2 The Multi-Agent Framework

To support cooperative goal achievement among a group of agents a delegation framework has been developed (Doherty and Meyer, 2007). It provides a formal approach to describing and reasoning about what it means for an agent to delegate an objective, which can be either a goal or a plan, to another agent. The concept of delegation allows for studying not only cooperation but also mixed-initiative problem-solving and adjustable autonomy.

By delegating a partially specified objective the delegee is given the autonomy to complete the specification itself. By making the objective more specific the autonomy is limited. If the delegated objective is completely specified then the agent has no autonomy when it comes to achieving the objective. By allowing agents and human operators to partially specify an objective mixed-initiative problem-solving is supported.

An agent is a reactive, proactive, and social entity with its own thread of control (Bellifemine, Caire, and Greenwood, 2007). Agents communicate with each other using the standardized agent communication language FIPA ACL (FIPA, 2002), which is based on speech acts. An agent provides a set of *services*. A service encapsulates a set of tasks that an agent can do in the form of speech acts that the service supports. A physical platform, such as a UAV, often hosts many different agents. Each agent is FIPA compliant and is implemented using the Java agent development framework JADE (Bellifemine, Caire, and Greenwood, 2007).

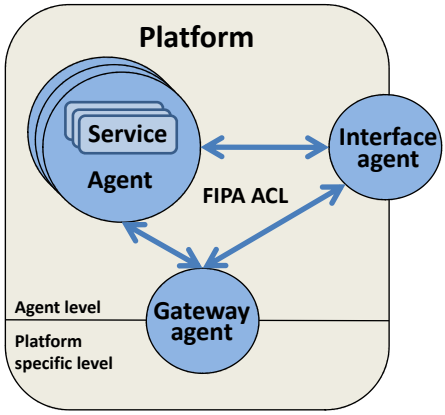


Figure 9.2: An overview of a platform in the delegation framework.

Each UAV platform has an agent layer consisting of a set of agents communicating using FIPA ACL and a layer with the platform specific functionalities (Figure 9.2). The interface between the two layers is the *Gateway Agent*, which provides a FIPA ACL interface to the platform specific level. In our UAV platform, where the platform specific software is implemented using CORBA, this involves invoking methods on different CORBA objects.

All communication between a platform and agents external to the platform goes through a single agent called the *Interface Agent*. The Interface Agent provides a single entry point to the platform which makes it possible to keep track of all communication, authenticate incoming messages, and perform access control to the platform.

A service can either be public, protected, or private. A public service can be used by any agent on any platform, while a protected service can be used directly within a platform but only indirectly through the Interface Agent by an agent on another platform. A private services can only be used by agents on the same platform.

To find services in the agent framework a *Directory Facilitator* (DF) is used. It is a database containing information about the available services such as what agent provides the service. There is a local Directory Facilitator on each platform which keeps track of the protected and private services of the platform and a global Directory Facilitator for keeping track of the public services in the multi-agent system.

9.3.3 DyKnow Federation Components

A platform taking part in a DyKnow federation should have three components: A DyKnow federation service, an export proxy, and an import proxy. A DyKnow federation service is a protected service which allows a local DyKnow instance to

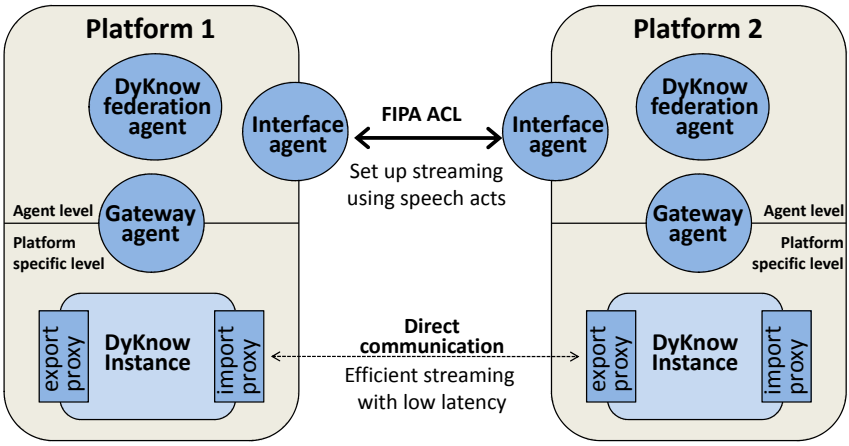


Figure 9.3: An overview of the components of a DyKnow federation.

take part in a DyKnow federation. It is protected to make all external requests go through the Interface Agent, which should be the single point of communication with other agents. The export and import proxies are used to mediate streams through direct communication between two DyKnow instances. Apart from these DyKnow federation specific components, the federation framework also uses the Interface and Gateway Agents from the agent framework. The Interface Agent is used to communicate with other platforms and the Gateway Agent is used to access the local DyKnow instance. Agents communicate using the FIPA ACL while two DyKnow instances communicate directly through the export and import proxies after setting up a stream through the DyKnow federation service (Figure 9.3).

To make a DyKnow instance available to other platforms it must be integrated in the agent framework. This is done in three steps:

1. By implementing the DyKnow federation service in an agent,
2. by extending the Interface Agent to provide the DyKnow federation service since the service is protected, and
3. by extending the Gateway Agent to allow the DyKnow federation service to access the local DyKnow instance.

One important issue is how to refer to information among platforms. A DyKnow instance will contain a set of labeled stream generators. The easiest approach would be to use these labels directly. One problem with this approach is that the agent level then must know what labels each of the other platforms have in their local DyKnow instances. This is not a major issue if all platforms are built by the same people, but in a more general setting this would not be easily done. A more feasible approach is to agree on a set of labels with a certain meaning among a group of agents called *semantic labels*. These semantic labels can then be translated by each agent to local DyKnow labels using whatever procedure necessary.

For example, a group of UAVs could agree that the semantic label heli-position is used to refer to their own position. This is a first step towards introducing a common ontology of information among a group of agents. The benefits are that each group of agents can use their own set of semantic labels, with a meaning they have agreed upon, and that the labels in the local DyKnow instances are isolated from each other.

The DyKnow Federation Service

The DyKnow federation service is responsible for supporting the finding and sharing of information among local DyKnow instances. An agent which implements the DyKnow federation service on a platform is called a DyKnow Federation Agent. This agent is registered in the local Directory Facilitator to make this platform available for federation. If an agent wishes to make a request to a DyKnow Federation Agent on a particular platform, it sends this request to the Interface Agent on that platform, which forwards the request to the DyKnow Federation Agent. The DyKnow Federation Agent is then responsible for fulfilling the request by using the Gateway Agent to access the local DyKnow instance.

Since we are working with agents which have their own agendas it is not guaranteed that they will accept all requests. However, unless there is a special reason they will in general accept all requests that they have the capability to perform.

The following messages can be sent to a DyKnow federation service:

1. Request a create stream multicast from a semantic label, a sample period, a maximum delay, a start time, an end time, and a set of receivers. This request should create a fluent stream for the semantic label satisfying the given sample, delay, and duration constraint. This fluent stream should then be exported to all the receivers.

   For example, UAV A could send a create stream multicast request for the semantic label heli-position to UAV B with a sample period of 2 seconds and UAVs C and D as receivers. If the request is accepted by UAV B, then it will export a fluent stream with its helicopter position sampled every 2 seconds to UAVs C and D.

2. Query the latest value, the value at a particular time-point, or all the values between two time-points for the fluent stream associated with a semantic label. The answer should be returned to the sender in an inform message.

   For example, if a UAV would like to know the position of another UAV then it could query the other UAV about the latest value of the stream associated with the semantic label heli-position. If the query is accepted then the answer will be looked up in the local DyKnow instance and returned in an inform message.

Each request can be either *local* or *global*. If a local request is made to a DyKnow federation service then only the local DyKnow instance on the platform will be queried. If the request is global then the federation service will query all

other platforms as well. This is done by the DyKnow federation service asking the Directory Facilitator about all agents providing the DyKnow federation service and forwarding the request to each of them. It will then aggregate the result and send it to the original requester.

The DyKnow Gateway

The DyKnow Gateway interface extends the Gateway Agent to allow the DyKnow federation service to access the local DyKnow instance.

On the platform specific level there will be four CORBA servers, one for each of the interfaces (DyKnow Gateway, Export Proxy, and Import Proxy) and one DyKnow location which makes the imported streams available to the local DyKnow instance. The DyKnow Gateway will be called by the Gateway Agent, while the import and export proxies are not called from the agent level.

The interface that an export proxy should implement is:

- `create_stream_multicast`(s, f, t, p, d, u), where s is a semantic label, f, t, p, and d are the from, to, sample period, and delay arguments used to create a fluent stream policy, and u is the set of receiving platforms.
- `create_stream_unicast`(s, f, t, p, d, w), where s is a semantic label, f, t, p, and d are the from, to, sample period, and delay arguments used to create a fluent stream policy, and w is the receiving platform.

The method `create_stream_multicast`(s, f, t, p, d, u) is used to start exporting the stream associated with the semantic label s with the constraints from f, to t, sample period p, and delay d to the platforms u. This method calls the DyKnow Gateway interface method `create_stream_unicast`(s, f, t, p, d, w) for each of the platforms w in u. The `create_stream_multicast` method is called by the Gateway Agent in response to a `create-multicast` FIPA ACL message.

The `create_stream_unicast`(s, f, t, p, d, w) method does the following:

1. Translates the semantic label s to a local DyKnow label l;
2. creates a fluent stream policy p from the constraints from f, to t, sample period p, and delay d;
3. uses the policy p and the label l to create a new fluent stream in the DyKnow instance; and
4. exports the new fluent stream by invoking the `start_exporting`(l, s, w) method in the local export proxy.

For example, for platform 1 to distribute heli-position to platform 0 with the constraints sample every 1 second and max delay 1 second the call would be `create_stream_unicast(heli-position, 0, 0, 1, 1, 0)`, if the local time-scale is seconds. Assume the local DyKnow instance on platform 1 has a stream generator called heli_pos which can generate fluent streams of the position

of platform 1 and that the label for the exported fluent stream is export_heli_pos. Then the policy for the created fluent stream is "**sample every** 1, **max delay** 1".

Export Proxy

An export proxy is a component used by a platform to export one or more streams. To export a stream an internal subscription is made by the proxy which then makes the stream available to other platforms in an implementation specific way. It would also be possible for the export proxy to reuse the DyKnow middleware to implement this functionality, but it might not be the best choice in all situations.

The interface that the export proxy should implement is:

- `start_exporting`(l, s, w), where l is a label, s is a semantic label, and w is a receiver.

The `start_exporting`(l, s, w) method in the ExportProxy interface creates a subscription to the fluent stream generator l. Each time a new sample v is pushed on the new fluent stream the `push`(m, s, v) method is called on the ImportProxy object on the receiving unit w, where m is the unit number of the sending platform. The unit number of a platform is its unique identifier. For example, to continue the example above each time a new sample is added to the export_heli_pos fluent stream the `push` method on the import proxy of platform 0 is called with the arguments platform 1, the semantic label heli-position, and the new sample.

Special care has to be taken when implementing the `start_exporting` call since the other platform might no longer be available or it might be busy and not accept the call directly. One approach is to make the export proxy multi-threaded with one thread for each remote platform. In this way no other platforms will be affected by a stop in the communication.

Import Proxy

An import proxy is a component used by a platform to import one or more streams. Each imported stream will be provided as a source in the local DyKnow instance. How the stream is imported is an implementation detail which must be coordinated with the export proxy. Different pairs of proxies can use different methods to communicate.

The interface that an import proxy should implement is:

- `push`(m, s, v), where m is the sender, s is the semantic label, and v the sample.

The `push`(m, s, v) method in the ImportProxy interface will translate the semantic label s to a label l, and add the sample v to the local stream generator associated with the label l. If this is the first sample for this semantic label then a new source is created and its stream generator is associated with the label l.

9.3.4 DyKnow Federation Functionalities

Adding and Removing Nodes

To make a node available for federation the Interface Agent of that node has to register its DyKnow federation service in the Directory Facilitator. When this is done the node is available, but no information is yet shared between the node and other nodes. To leave a DyKnow federation it is enough to unregister the DyKnow federation service. It is possible to allow active streams to and from the node to remain even after the node leaves the federation since the proxies talk directly with each other when the streaming has been set up.

Query for Information

If a platform needs a particular piece of information, knows its semantic label, and knows which platform can provide the information then a query can be sent to the Interface Agent of that platform with the semantic label as the argument. Using this method a platform could ask for the latest value of a stream, the value at a particular time-point, or all the values between two time-points.

If a platform only knows the semantic label of the information, but not which platform is hosting the information, then it has to make a global request. This will cause the DyKnow federation service to query all platforms providing a DyKow federation service for the semantic label. This could give any number of answers. If the DyKnow federation service gets more than one answer then it has to either select one of the values or merge them together.

To implement the first use case, explicit ask and tell to divide the monitoring area, this functionality would be used. The leader UAV would make a global request for the current value of the streams associated with the semantic labels max-speed, fuel, and so on.

Streaming Information

Setting up a stream from one platform to another is different from requesting a particular piece of information directly. Instead of sending something back, the agent receiving the request will set up an export proxy which will start streaming the information to the import proxy of the requesting agent.

When proxies are set up the platform that made the request can access the stream through a local fluent stream generator. From the point of view of the local DyKnow instance, the import proxy is another source of information, like a sensor.

This would be the main functionality required to implement the second use case, to provide continuous information about tracked vehicles from one UAV to another. The UAV receiving the stream would then have to fuse this stream with its own stream of car estimations in order to do the qualitative spatial reasoning and chronicle recognition.

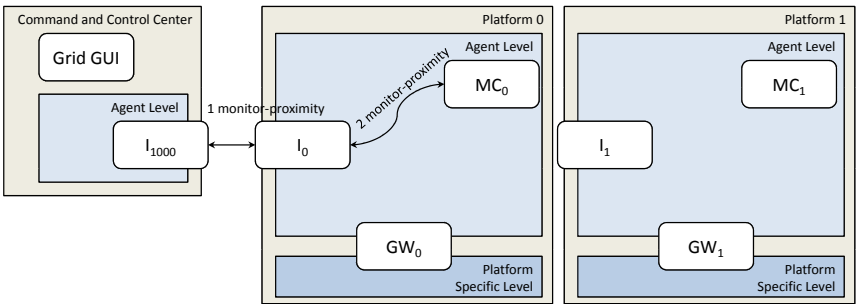


Figure 9.4: The task of monitoring the proximity is delegated from the command and control center to platform 0.

9.4 Implementing the Proximity Monitoring Scenario

This section describes how the DyKnow federation framework can be used to implement the proximity monitoring functionality introduced in Section 9.2.1 on page 181. The description of the implementation is divided into two parts, the agent level and the platform specific level. The two layers are separated by the Gateway Agent.

9.4.1 Implementing the Agent Level

The proximity monitor functionality is provided by a monitor coordination service which is implemented by a *Monitor Coordination Agent*. It coordinates the collection of information, sets up the monitoring, and informs the other UAVs in case the proximity constraint is violated. The collection of information and the monitoring is done using the local DyKnow instance which is accessed through the Gateway Agent. Since the Monitor Coordination Agent also implements the DyKnow federation service no separate DyKnow Federation Agent is used.

The proximity monitoring functionality is invoked by delegating the task of `(monitor-proximity)` to a platform which has registered a monitor coordination service (Figure 9.4). In our example it is the command and control center which delegates this task to platform 0. The Interface Agent on platform 0, I_0, will accept the delegation and call its local Monitor Coordination Agent, MC_0, and request it to perform the monitoring task.

To perform the monitoring task MC_0 checks what other agents have registered with the agent framework and what semantic labels are required to evaluate the monitor formula. Using this information it can delegate the task of exporting the streams associated with the set of semantic labels to each registered platform (Figure 9.5). To delegate this task to each registered platform MC_0 has to go through the Interface Agent I_0 which does the actual delegation (step 1 in Figure 9.5). In our example there are two platforms and the semantic labels needed are heli-position and heli-altitude.

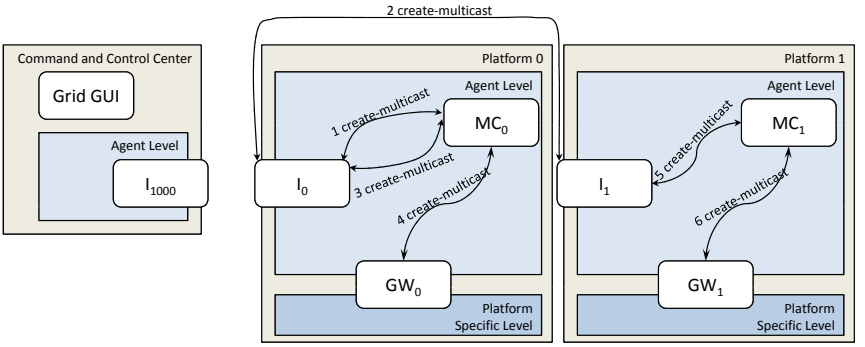


Figure 9.5: The necessary information sharing is set up by the Monitor Coordination Agent MC_0.

It is not enough to only have the semantic labels. The constraints specifying the properties of the exported streams are also required. In this implementation we will sample the values for each semantic label every second and accept a delay of one second. No duration constraint is placed on the streams.

Therefore, platform 0 delegates to platform 1 the task of multicasting heli-position and heli-altitude to platform 0 with the constraints sample every 1 second and max delay 1 second (step 2 in Figure 9.5). The actual message is:

```
(create-multicast
  semantic-labels: (sequence heli-position heli-altitude)
  sample-period: 1.0 delay: 1.0
  to-units: (sequence 0))
```

Platform 0 will also set up a multicast to distribute its own position and altitude to platform 1 (step 3 in Figure 9.5).

To set up a multicast a Monitor Coordination Agent sends a request to the Gateway Agent to call the DyKnow instance which does the actual work (step 4 and step 6 in Figure 9.5).

The third and final step is to start the monitoring (Figure 9.6). This is done by a Monitor Coordination Agent by delegating the task of creating a proximity monitor to each registered platform including itself (steps 1–3 in Figure 9.6). The actual message is: `(create-monitor name: proximity)`. To set up a local monitor, a Monitor Coordination Agent sends a request to the Gateway Agent to call the local DyKnow instance which does the actual monitoring (steps 4 and 6 in Figure 9.6). How this is done is described in Section 9.4.2.

If a proximity monitor is violated then the Monitor Coordination Agent will send a message to the other platform which contributed to the violation in order to initiate an avoidance maneuver.

During the monitoring the Monitor Coordination Agent MC_0 is responsible for checking if new platforms register with the agent framework. If a new platform is registered then the information sharing and monitoring tasks are delegated to

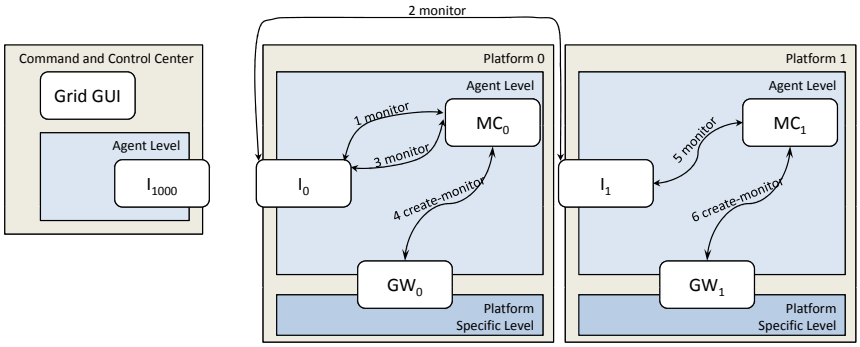


Figure 9.6: The proximity monitoring is started by the Monitor Coordination Agent MC_0.

it by MC_0. The new platform will then have to delegate the task of streaming updates related to the necessary semantic labels to each of the platforms in the agent framework, since they do not keep track of new platforms themselves. By implementing a broadcast functionality instead of a multicast it would have been the responsibility of each platform to detect new platforms.

9.4.2 Implementing the Platform Specific Level

To set up a monitor the method `create_named_monitor`(n) in the DyKnow Gateway interface is called by the Monitor Coordination Agent. The actual monitor formula is implicit to make the knowledge of it local to the DyKnow Gateway implementation. Currently the monitor formulas are hard coded and referred to by name, but this is not a fundamental limitation and it is easy to add new names.

If the name is `proximity` then the `create_named_monitor` method adds the formula $\Box(\text{xy\_dist}[h0, h1] \geq 15.0 \wedge \text{z\_dist}[h0, h1] \geq 10.0)$ to the local execution monitoring service. The formula says that the distance in the xy-plane between platform 0 and 1 must always be greater than 15 meters and the distance in the z-plane must always be greater than 10 meter. The formula uses the fluent streams xy_dist[h0,h1] and z_dist[h0,h1] which are computed from the position and altitude respectively by computational units. Each of the computational units takes one stream which is produced locally and one stream which is produced on the other platform, synchronizes the samples, and then computes the distance between the platforms.

To evaluate the proximity monitoring formula a fluent stream must be created for each of the features used in the formula. This is currently done by the import proxy. When the import proxy on a platform is created it will create a source in the import location for each of the semantic labels it knows. It will also create the necessary computed fluent streams and state streams to evaluate the formula. For example, in the proximity monitoring example each platform would create a source for the helicopter position and the helicopter altitude of the other platform.

When the import proxy has been created the platform is ready to receive position and altitude information from the other platforms and evaluate whether they are coming too close or not. As soon as the goal of monitoring the proximity is given the sharing of the information is set up and the actual monitoring can be started. The scenario has been tested both in simulation and live flight tests.

9.5 Summary

A DyKnow federation framework for information integration in a distributed multi-node network of UAVs has been presented. This type of framework is required to develop complex multi-agent systems where agents have to cooperate to solve problems which are beyond the capability of any individual agent. The framework allows agents to share and merge information to provide more complete and accurate information about the environment.

The federation framework is an extension of DyKnow which integrates DyKnow with a FIPA compliant multi-agent framework. The extension allows an agent to share parts of its local DyKnow instance with other agents in a DyKnow federation. The basic interaction and sharing is made on an agent level using the standardized FIPA ACL agent communication language. To increase the efficiency, direct communication is supported for continuous streaming of information between nodes. In either case the federation is used to find information and to set up the distribution.

The main contribution of this chapter is how the versatile and useful knowledge processing middleware framework DyKnow can be extended to cover an even larger set of issues and allow it to integrate not only sensing and reasoning on a single platform, but also finding and sharing of information among multiple platforms. This chapter provides the motivation, requirements, and initial design of such an integrated framework which is being implemented and tested on our existing UAV platforms.

Part IV

Conclusions

Chapter 10

Relations to the JDL Data Fusion Model

10.1 Introduction

To improve the quality of the available information about a specific subject it is often useful to combine and refine information from many different sources. It can be that no single source contains all the required information. For example, to detect traffic patterns it might be necessary to use both a geographical information system to get the details about the current road system and a suit of sensors providing information about the vehicles driving on the road system. It could also be the case that by combining information from many sources the uncertainty in the information can be reduced. For example, robots often combine the odometry information from an inertial measurement unit with the global position estimation provided by a GPS to estimate their current position.

How to combine and refine information from multiple sources is often called the *fusion problem*. Depending on the type of sources and the type of output from the process one often adds a specific prefix, referring to sensor fusion, data fusion, information fusion, and so on. In this chapter we will use the term fusion without any prefix to indicate the general nature of our approach.

One definition of data fusion which is provided by the JDL Data Fusion Subgroup (1987) is:

> A process dealing with the association, correlation, and combination of data and information from single and multiple sources to achieve refined position and identity estimates, and complete and timely assessments of situations and threats, and their significance. The process is characterized by continuous refinements of its estimates and assessments, and the evaluation of the need for additional sources, or modification of the process itself, to achieve improved results.

Since the fusion problem is very general there is a need to provide a common

framework for talking about different approaches to the different parts of the fusion problem. One such framework is the JDL Data Fusion Model (Llinas et al., 2004; Steinberg and Bowman, 2001). It is a functional model which is often considered to be the de facto standard fusion model. The JDL Data Fusion Model provides a common framework where fusion functionalities are divided into a number of different abstraction levels described in the next section.

Usually the fusion problems on each of the functional levels are solved using different methods and approaches. Many interesting applications, such as monitoring traffic and assisting emergency services, require fusion on many different levels to be used in a single application. Therefore, there is a need to integrate these different approaches. We believe that DyKnow provides a framework with the appropriate concepts and mechanisms to integrate different existing partial solutions to the fusion problem into more advanced applications. It is important to realize that DyKnow does not solve the different fusion problems involved, but rather provides a framework where different specialized fusion algorithms can be integrated and applied.

10.2 The JDL Data Fusion Model

The JDL Data Fusion Model is the most widely adopted functional model for data fusion. It was developed in 1985 by the U.S. Joint Directors of Laboratories (JDL) Data Fusion Group (White, 1988) with several revisions proposed (Blasch and Plano, 2003; Llinas et al., 2004; Steinberg and Bowman, 2001). The purpose of the model is according to Steinberg and Bowman (2001) to "facilitate understanding and communication among acquisition managers, theoreticians, designers, evaluators, and users of data fusion techniques to permit cost-effective system design, development, and operation".

The data fusion model originally divided the data fusion problem into four different functional levels (White, 1988). Each of the first three levels builds on the previous level by making the information more abstract. The fourth level supports the dynamic adaptation of the fusion process itself. Later a level 0 performing sub-object assessment (Steinberg and Bowman, 2001) and a level 5 performing user refinement (Blasch and Plano, 2003) were introduced. The levels 0-4 as presented by Steinberg and Bowman (2001) are shown in Figure 10.1 and described below.

- **Level 0** - *Sub-Object Data Assessment*: Estimation and prediction of signal- or object-observable states on the basis of pixel/signal-level data association and characterization.
- **Level 1** - *Object Assessment*: Estimation and prediction of entity states on the basis of inferences from observations.
- **Level 2**- *Situation Assessment*: Estimation and prediction of entity states on the basis of inferred relations among entities.
- **Level 3** - *Impact Assessment*: Estimation and prediction of effects on situations of planned or estimated/predicted actions by the participants (e.g.,

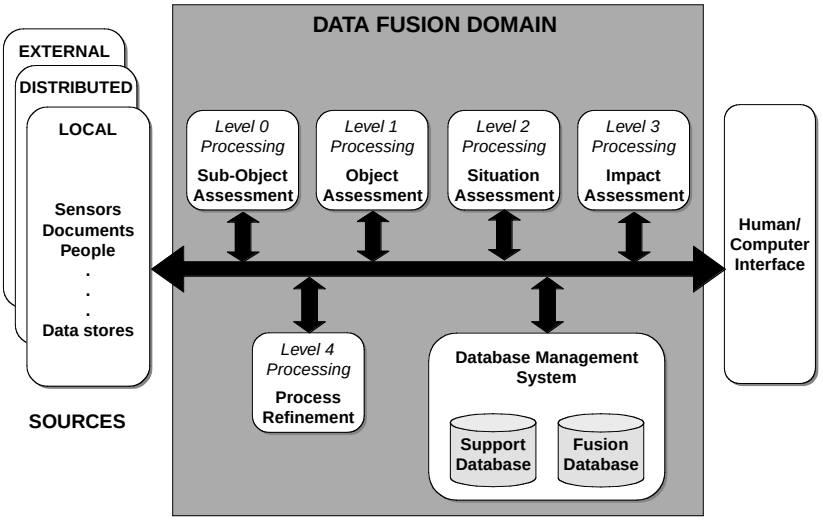


Figure 10.1: Revised JDL data fusion model from Steinberg and Bowman (2001).

assessing susceptibilities and vulnerabilities to estimated/predicted threat actions, given one's own planned actions).

- **Level 4** - *Process Refinement*: Adaptive data acquisition and processing to support mission objectives.

The rest of this chapter describes how DyKnow can support the functionalities on each of the levels in the JDL Data Fusion Model. This shows that DyKnow provides ample support for fusion on all levels of abstraction.

10.3 JDL Level 0 – Sub-Object Data Assessment

On level 0 signals and sub-object features are fused. This could for example involve detecting the presence of a signal and estimating its state. An example is to recognize a feature in an image. One purpose of fusion on this level is to reduce the noise and uncertainty in the signals in order for the higher levels to get the best possible input to work with.

All fusion applications developed using DyKnow are structured as a set of knowledge processes taking streams as inputs and generating new streams. Any fusion algorithm that can be described as a process taking streams as input and generating streams as output can be integrated in the DyKnow framework as a knowledge process.

Since a stream can be seen as a digital signal where each sample is the value of the signal at a particular time-point there is a direct mapping between digital signal processing and stream processing. For example, to fuse two sub-object level

signals they first have to be integrated in the DyKnow framework by making them available as sources. When the signals are made available any knowledge process, such as a fusion process, can create and subscribe to fluent streams corresponding to the signals. The output of the fusion process would be a third stream containing the fused result.

An example of a level 0 functionality in the traffic monitoring application is when the image processing system detects the presence of a car-like object in an image. If the sequence of color images and the sequence of thermal images are seen as two signals then they could be fused on level 0 producing a combined color and thermal image signal.

The sub-object features are used mostly at level 1 to estimate object states.

10.4 JDL Level 1 – Object Assessment

On level 1 sub-object data are fused into coherent object states. The purpose is to detect individual objects, to estimate their current state, and to generate tracks estimating their states over time.

A traditional level 1 fusion approach is to use a Kalman filter to estimate the current position of a robot by fusing odometry information from an inertial measurement unit with global position estimations provided by a GPS. In DyKnow this functionality would be provided by a computational unit implementing the Kalman filter and fluent streams modeling the odometry, GPS, and position information.

Level 1 is usually partitioned into four functions (Hall, 1992): Data alignment, data association, tracking, and identification. Data alignment tries to put data from different sources into a common frame of reference, such as a common coordinate system. Data association then tries to group the data into clusters where each cluster only contains data about a single object. The tracking functionality then tries to estimate the object state, mainly the position and velocity of each of the objects. The focus on position and velocity indicates the history of the research area in the military target tracking community. Finally, identification tries to classify the target and to extract more information about it besides its current location.

DyKnow provides support for all four of these functionalities, with a special emphasis on data association and identification. As usual, almost any functionality can be realized as a knowledge process which means that DyKnow does not prevent any existing approaches from being used.

Data alignment in the temporal domain is supported in DyKnow through the state extraction mechanism described in Section 7.8. It allows two or more streams to be aligned in time by creating a state containing an element from each of the streams, where all the elements in each state is valid at the same time. Aligning streams in time is especially important in distributed applications where the data can be generated by sources on different platforms without any synchronization. Example 7.8.1 on page 118 provides an example where the data generated by two sensors is synchronized in time. An appropriate fusion algorithm can then be applied to the resulting state stream.

The object linkage structures described in Section 8.5 provide sophisticated support for data association, tracking, and identification. Anchoring is in itself a form of data association where sensor data is associated with symbols. Object linkage structures provide support for both associating sensor data with objects and incrementally classifying these objects. To use the functionality it is enough to define a set of classes which are of interest and then describe the conditions in a link specification for when to associate an instance of a class with an instance of another class.

For example, in the traffic monitoring scenario there are three classes: Vision object, world object, and on road object. Vision objects can be linked to world objects, hypothesizing that the entity seen in the image is a physical object in the world. A vision object is transformed to a world object by converting the position coordinates from an image-centered coordinate system to a common "world" coordinate system. This is in fact another example of data alignment. If a world object is observed to be on the road system, as defined by a geographic information system, then it is hypothesized that the world object is actually an on road object. The data extracted by the image processing system is now being associated first with a world object and then an on road object. As long as this chain of associations is maintained and the vision object is tracked, the world object and then the on road object will be updated each time a new picture is taken and the vision object is updated.

In DyKnow, streams from level 1 mainly interact with level 2 by providing coherent object states for computing and detecting situations. Level 3 is also very important since it is responsible for checking the hypothetical object linkage structures by continually monitoring the impact of new observations on the current hypotheses. Since the computations on level 1 can be time consuming, the interactions with level 4 is also important in order to maintain a steady update of the most important fluent streams for the moment. Level 4 can for example change the sampling rate, increase the amount of allowed delay, or remove less important fluent streams.

10.5 JDL Level 2 – Situation Assessment

On level 2 relations between objects detected on the previous level should be detected and put in a larger context to estimate the current situation. A situation often involves a number of objects satisfying a set of spatial and temporal constraints. Describing such situations and detecting instances of the situations is an important functionality at this level.

DyKnow has two different mechanisms for describing and detecting situations. Both the metric temporal logic introduced in Section 7.5.1 and the chronicle formalism described in Section 8.2 can be used to recognize complex temporal relations between objects and their attributes.

In the traffic monitoring application a reckless overtake is a typical example of a situation which is of interest to detect. To describe and detect reckless overtakes a chronicle can be defined using qualitative spatial relations such as

beside(car_1, car_2), close(car_1, car_2), and on(car, $road$). A chronicle could add metric temporal constraints to the qualitative spatial constraints described by the spatial relations. Since the temporal constraints can be metric it is possible to combine both qualitative and quantitative reasoning as well as spatial and temporal reasoning. Given a chronicle definition and streams of car object states generated using level 1 functionality it is possible to recognize instances of the reckless overtake chronicle.

Combining the situation assessment functionalities with the object assessment functionalities it is possible to provide support for a concept such as "the set of all cars that have been observed making reckless overtakes in the last 30 minutes". To describe and maintain sets of related objects, where the set changes over time as the properties of the objects changes, object linkage structures can be used. By using the establish and maintain condition of a link declaration to describe the conditions for when a car should be linked to a reckless car a stream of all reckless cars can be created. The link function as a classification procedure which collects all objects which satisfy the establish condition and keeps them as members as long as the maintain condition is not violated (see Section 8.5).

To maintain the set of cars that have been observed making reckless overtakes, functions on JDL levels 0 to 3 have to work in concert. To detect cars sub-object features have to be extracted from color and thermal images. These sub-object features then have to be classified as belonging to a car and associated with a car symbol. From the stream of car object states it is then possible to detect reckless overtakes as a complex spatio-temporal relation using chronicle recognition. When a car has been observed making a reckless overtake it is automatically added to the class of reckless cars, by linking the car object to a reckless car object. However, to stay a member the car has to be observed making a reckless overtake every 30 minutes or else it will be removed from the class. To monitor this condition a metric temporal logical formula is used. This is an example where a complex concept can be supported through the interaction of the different functionalities provided by the DyKnow framework.

Another mechanism which can be used to collect information about a situation is state generation. Collections of object states can be aggregated into states in order to synchronize them to a coherent situation, just as collections of fluent streams can be collected into states.

Apart from the input provided by streams at level 1, the interactions of level 2 are mainly with level 3 where streams representing complex situations can be used to maintain object linkage structures as well as create new hypotheses.

10.6 JDL Level 3 – Impact Assessment

On level 3 the impact of the object and situation assessment on the current actions and plans of an agent is assessed. A typical task on this level is to determine if the current plans are working as expected or if there are threats or opportunities which should be considered.

One approach to continually assessing the impact of the current situation is to monitor a set of conditions describing those situations which have an impact. If one of the monitors is triggered then the system has to react to that event. We have for example described how DyKnow monitors the current classification hypotheses by monitoring the maintain conditions of links. If such a condition is violated then the link is removed and the two objects are no longer classified as representing the same physical object.

DyKnow supports the definition of complex monitors through the use of the complex event detection mechanisms described in the previous section. As shown in the example in the previous section, chronicle recognition and metric temporal logical formulas can be combined to support complex conditions to be monitored. Both mechanisms are efficient enough to perform this monitoring effectively.

Another example of monitoring the impact of the current situation on the current plans of an agent is execution monitoring as described in Section 7.5. By integrating the monitoring conditions in the planning domain it is possible for the planner to take the conditions into consideration when creating the plan. The generated plan contains the conditions, expressed in logic, to be monitored during the execution of the plan. If a condition is violated the execution system is given the possibility to react to the violation and perform a recovery operation. It is also possible to call the planner again to generate a new plan to take the new situation into consideration.

Level 3 interacts with both level 1 and level 2 since the streams produced on those levels are the ones used as input to impact assessment. The detection of violations of monitored constraints will lead to changes at the lower levels.

10.7 JDL Level 4 – Process Refinement

On level 4 the system should adapt the data acquisition and the processing to support mission objectives. In DyKnow, the whole process is described as a set of knowledge processes generating streams and refining the process corresponds to changing what streams are being computed. This is related to focus of attention issues where the most important streams should be computed while less important streams have to stand back in times of high loads.

The main tool for supporting process refinement and focus of attention is policies. A policy describes the desired properties of a stream. By changing the policies of the streams the load can be reduced. For example, consider an application where the GPS position is sampled at 50 Hz and there is a chain of processes which are all dependent on the position. If the latency when updating all the processes in the chain is larger than 20 milliseconds then the application will lag further and further behind. To manage the situation it is possible to change the policy of the position stream to for example sample the GPS at 25 Hz instead. This single change in a policy will affect the whole application since the input frequency is reduced by 50%. If the overload of the system is only temporary it is possible to change back to the original policy later.

To detect that a process needs refinement the functionality at level 3 can be used to set up the desired monitors.

Level 4 interacts with all the other levels since it controls the context within which those are being computed by controlling the policies.

10.8 Summary

This chapter has presented a high level view of how DyKnow provides support for the functionalities on the different levels in the JDL Data Fusion Model.

As long as a fusion component can be described as a process which takes streams as input and generates streams as output it can easily be integrated in the DyKnow framework. This makes it possible to use DyKnow to integrate existing partial solutions to the fusion problem into fusion applications.

One important aspect is that DyKnow allows functionalities on the different JDL levels to interact in a seamless manner. This chapter has shown how the different parts of DyKnow can work in concert to produce information and knowledge on all levels of abstraction. Starting from primitive sensor data the end result of the processing is highly qualitative information and knowledge.

The conclusion is that DyKnow provides an integrated framework where fusion applications using functionalities on all the JDL levels can be supported.

Chapter 11

Related Work

11.1 Introduction

In this chapter stream-based knowledge processing middleware and DyKnow are put into perspective by comparing them to other related approaches.

Section 11.2 relates DyKnow to distributed real-time databases. We will argue that distributed real-time databases systems do not by themselves satisfy any of the requirements from the introduction. Rather, they address relevant but different research issues.

Section 11.3 relates DyKnow to existing agent and robot control architectures. Even though DyKnow is not such an architecture they do partly address the same issues. We will argue that DyKnow could be used by these architectures to further improve them.

Section 11.4 focuses on some frameworks providing general support for integrating sensing and reasoning. We will argue that even though there exist many interesting frameworks none of them satisfy the requirements well enough. For example, none of the frameworks have a formal basis and most of them do not handle time as well as DyKnow. With general support we mean that a system does not prevent any of the requirements introduced in Section 1.2.1 on page 11 from being met. However, the explicit support for the requirements often wildly differs.

11.2 Distributed Real-Time Databases

What separates a real-time database from a traditional database is that transactions may be associated with timing constraints (Ramamritham, Son, and Dipippo, 2004). These timing constraints must be satisfied together with the normal consistency constraints. Examples of timing constraints are completion deadlines, start times, and periodic invocations. Since timing properties are essential for real-time databases it is necessary to make query and scheduling algorithms aware of time. Non-real-time databases generally try to minimize the average response time by

maximizing the resource utilization while real-time databases will first maintain timing constraints and only in second place consider average response times and resource utilization.

An important issue in real-time data management systems is data consistency and freshness. Not only should a database meet all the timing requirements, it should also use the most current and up to date information (Gustafsson, 2007). Each time new information has been received, all the components dependent on this information have to be updated. Since these updates can be expensive and might lead to an increasing number of missed deadlines, a database has to make a decision whether new information should be accepted or not. This allows the database to make a trade-off between data freshness and deadline miss ratio (Kang, Son, and Stankovic, 2004).

Managing the quality of service (QoS) is another important issue. A QoS specification could for example express the desired requirements on data freshness, deadline miss ratio, and the trade-off between them. Traditional approaches rely on knowing the worst-case execution times of the system (Buttazzo, 1997). Recently, a new class of approaches based on feedback control have been developed in order to handle uncertain and time-varying load (Amirijoo, 2007; Lu et al., 2002). In these approaches, the current load and execution times are used to regulate the admission control and the scheduling of a system. Therefore there is no need to reserve resources for the worst case. In this way resources are used more effectively.

The complexity of these issues increases further when considering distributed real-time databases (Shanker, Misra, and Sarje, 2008). Important issues with respect to distribution are replication and consistency among distributed nodes and distributed real-time commit protocols that satisfy timing constraints for distributed real-time transactions in the face of unpredictable communication delays and multiple concurrent transactions. Since DyKnow does not replicate streams or support atomic transactions these issues are avoided. DyKnow still has to manage unpredictable communication delays, but these are assumed to be taken into account in the delay constraint in the policy of a stream.

A distributed real-time database system does not prevent any of the requirements from the introduction from being satisfied. Neither does it by itself provide very much support for them. However, the techniques developed within this community are relevant and useful when designing and implementing knowledge processing middleware with real-time guarantees. Especially the issues of data freshness and satisfying timing constraints are important. Currently DyKnow focuses on reducing the average response time without providing real-time guarantees. As future work, it would be interesting to apply the research on admission control and scheduling to DyKnow to provide basic real-time guarantees and graceful degradation in the face of temporary overloads. Especially the work on using feedback control to manage real-time quality of service guarantees seems appropriate.

11.3 Agent and Robot Control Architectures

An agent architecture defines how to structure the functionality of an agent or robotic system as a number of subsystems and how these subsystems interact. An architecture provides a way of managing the growing complexity of building robotic systems by embodying well-defined concepts which enable the effective realization of systems to meet high-level goals (Coste-Maniere and Simmons, 2000). The focus is usually on how to control an agent to achieve goals. Important issues are how to decompose the control problem and how to manage the trade-off between reactive stimulus-response behavior and deliberative goal-directed actions. Depending on this trade-off a distinction is made between reactive, deliberative, and hybrid architectures.

A reactive architecture couples sensors more or less directly to actuators through some form of stimulus-response mechanism. One common trait of these architectures is that they usually do not create or require any models of the environment. Well-known examples of reactive architecture are Brooks' Subsumption Architecture (Brooks, 1985, 1991) and different forms of behavior-based architectures (Arkin, 1998).

A deliberative architecture, on the other hand, focuses on the goal-directed deliberation on what actions to perform in order to achieve the goals of an agent. These architectures are dependent on having models of the environment which are used to reason about the effects of different actions. Usually these architectures use a Sense-Plan-Act control loop where an agent first collects all the information it can from its sensors, then generates a plan to achieve its goals, and finally executes the plan.

Hybrid architectures try to combine the fast response of reactive architectures with the goal-directed nature of deliberative architectures. This has been a very active area and there are many architectures available (for example Arkin (1998); Atkin et al. (2001); Bonasso et al. (1997); Pell et al. (1998)).

Since deliberative and hybrid approaches often use symbolic reasoning they need to bridge the gap between the information available from sensors and the input required for reasoning about the environment. This means that one purpose of a deliberative or hybrid agent architecture is, in a sense, to describe how an agent bridges the sense-reasoning gap with respect to action. Existing agent architectures have mainly focused on integrating actions on different levels of abstraction, from control laws to reactive behaviors to deliberative planning. It is often mentioned that there is some form of parallel hierarchy of more and more abstract information extraction processes or that the deliberative layer uses symbolic knowledge (Andronache and Scheutz, 2003; Atkin, Westbrook, and Cohen, 2001; Barbera et al., 2003; Ingrand et al., 2007; Konolige et al., 1997; Lyons and Arbib, 1989; Rotenstein et al., 2007; Scheutz and Kramer, 2006; Shapiro and Ismail, 1998; Volpe et al., 2001), but only a few of these approaches are described in some detail. Two of these approaches will now be described in some detail.

11.3.1 The Hierarchical Agent Control Architecture

The Hierarchical Agent Control architecture (HAC) is a general toolkit for controlling agents (Atkin et al., 2001; Atkin, Westbrook, and Cohen, 2001). It supports action abstraction, resource management, and sensor integration. What makes HAC different is that it has a sensor hierarchy parallel to the hierarchy of actions. In the same way as complex actions are composed of simpler actions, complex sensors use the output of simpler sensors as input to provide more abstract or refined information. These sensors are not physical, but integrate and re-interpret sensor data from other sensors which may be physical.

Apart from the action and sensing hierarchies there is also a context hierarchy which consists of goals. A context provides assumptions for actions to operate within. This means that HAC separately manages the flow of control information, sensor information, and context. Each type of information corresponds to a separate hierarchy in HAC.

HAC supports the principle of supervenience, where higher level sensors integrate and interpret information from lower levels but without changing the lower level information. Lower level sensors provide information to the higher level ones but they do not control them. An advantage of this is that each level of the hierarchy can be treated independently without worrying about the interactions with the other levels.

In summary, HAC provides a uniform way of structuring the processing of sensor data which is appropriate in many situations. However, it lacks support for integrating information from distributed sensors and it does not have any support for the temporal aspects of information processing. Neither does it have any declarative description of the information processing so an agent can not reason about its own processing of sensor data.

11.3.2 4D/RCS

4D/RCS is a control system architecture inspired by a theory of cerebellar function (Albus, 1981, 2002; Barbera et al., 2003; Schlenoff et al., 2006). 4D refers to three dimensions of space and one dimension of time, and RCS stands for Real-time Control Systems. 4D/RCS models the brain as a hierarchy of goal-directed sensory-interactive intelligent control processes that theoretically could be implemented by neural nets, finite state automata, cost-guided search, or production rules.

4D/RCS tries to combine different knowledge representation techniques in a unified architecture. It consists of a multi-layered hierarchy of computational nodes each containing sensory processing, world modeling, value judgment, behavior generation, and a knowledge database (Figure 11.1 on the following page). The idea of the design is that the lowest levels have short-range and high-resolution representations of space and time appropriate for the sensor level while higher levels have long-range and low-resolution representations appropriate to deliberative services. Each level thus provides an abstract view of the previous levels. Each node may use its own knowledge representation and thereby supports multiple dif-

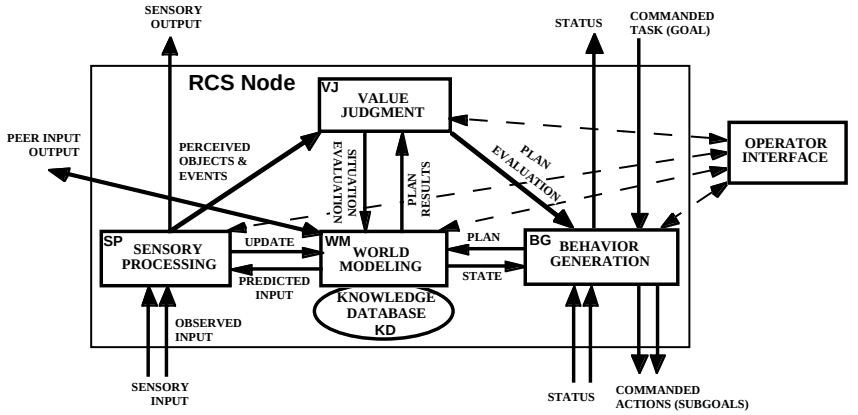


Figure 11.1: An overview of a 4D/RCS component (from Albus (2002)).

ferent representation techniques. However, the architecture does not, to our knowledge, address the issues related to connecting representations and transforming one representation into another.

4D/RCS represents procedural knowledge in terms of production rules and declarative knowledge in abstract data structures such as frames, classes, and semantic nets. Like DyKnow it also includes signals, images, and maps in its knowledge database, and maintains a tight real-time coupling between iconic and symbolic data structures in its world model. Some specific characteristics: Its focus on task decomposition as the fundamental organizing principle, its level of specificity in the assignment of duties and responsibilities to agents and units in the behavior generating hierarchy, and its emphasis on controlling real machines in real-world environments.

4D/RCS uses many different representations but the authors do not provide any details about how these representations are connected and controlled. For example, is there a specification of how the information in one representation is transformed into information in another representation. This leads to questions such as: How does a component know if a representation has changed in such a way that the component has to react?

11.3.3 Discussion

The gap between sensing and reasoning is present in all types of architectures, but it is especially important in hybrid architectures. The reason is that they must continually integrate information collected from sensors into symbolic representations required by deliberative functionalities. The existing architectures do not currently provide any general approaches to bridging this gap. Instead they use fixed solutions for particular problems such as navigating in unknown terrains (Ingrand et al., 2007; Thrun et al., 2006) and object recognition in combination with grasping

(Brenner et al., 2007). One reason for the lack of general approaches could be that it is very challenging to find generic solutions and it is often faster and cheaper to bridge the gap in robot and application specific ways.

As DyKnow focuses on information and knowledge processing and not on control it is orthogonal to most of the functionality provided by agent architectures. Since most existing architectures only have limited support for closing the sense-reasoning gap DyKnow could be used to support the development of such architectures. The sensor processing functionality they do provide could be integrated in DyKnow. The same is true for those specific approaches that have been developed to build particular models from sensor data, such as simultaneous localization and mapping (Montemerlo and Thrun, 2007) and transforming signals to symbols (Nii et al., 1988). They do not replace DyKnow, nor make it irrelevant. Instead they accentuate the need for DyKnow since there are so many disparate partial solutions which would benefit from being integrated. Therefore knowledge processing middleware, such as DyKnow, could become an important tool in the toolkit of agent architecture developers.

11.4 Robotics Middleware and Frameworks

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 an approach supports communication among distributed components it does not necessarily explicitly support information and knowledge processing.

There are a few surveys available (Biggs and Macdonald, 2003; Kramer and Scheutz, 2007; Mohamed, Al-Jaroodi, and Jawhar, 2008; Orebäck and Christensen, 2003). Of these, the survey by Kramer and Scheutz (2007) is the most detailed. It evaluates nine freely available robotic development environments according to their support for specification, platforms, infrastructure, and implementation. While it mainly focuses on software engineering aspects of these development environments we are more interested in how robotic frameworks support knowledge processing and bridging the gap between sensing and reasoning.

Before summarizing the support provided for knowledge processing by current robotics software framework we will give a high-level overview of a representative selection of existing frameworks.

11.4.1 ADE

ADE is an agent architecture development environment intended for the design, implementation, and testing of distributed robotic agent architectures (Andronache and Scheutz, 2004, 2006; Scheutz, 2006). It is based on APOC, a general and universal agent architecture framework which allows a wide range of agent architectures to be expressed and defined (Andronache and Scheutz, 2003).

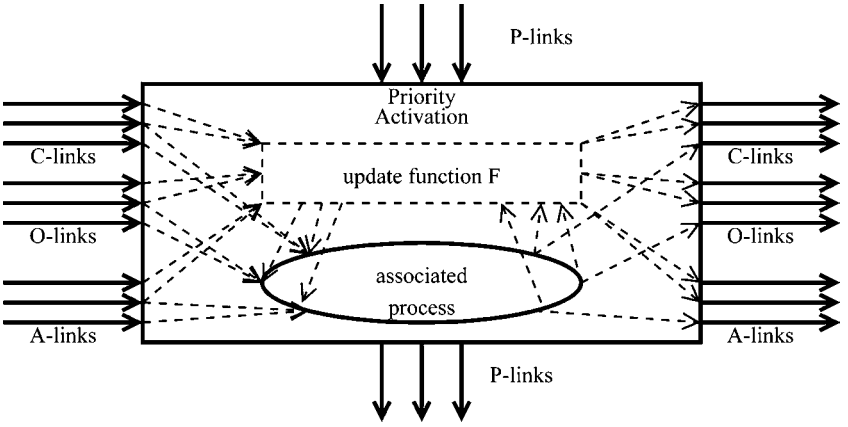


Figure 11.2: An overview of an ADE component (from Scheutz (2006)).

The APOC framework views agent architectures as networks of computational *components* connected by four types of *links* (Figure 11.2). Each component is an autonomous computational process with activation and priority values which are used for process management. A component can also be associated with a process which could for example be used to control a robot device or integrate an external function.

The links support different types of interactions among components in an architecture. The activation link allows components to send and receive messages. The observation link allows components to observe the state of other components. The priority link allows components to influence the execution of other components. The component link allows components to instantiate other components and connect to them via links. Each link type is defined formally.

By using different types of links many varieties and kinds of agent architectures can easily be created. An architecture instance is specified by an *architecture diagram* which describes how the components are connected using the different links. To provide limited support for managing resources it is possible to specify how many instances of a component may exist at the same time.

An APOC server provides mechanisms for instantiating, deleting, and updating components and links. Instantiated components and links are hosted by an APOC server. The update of components and links can either be synchronous or asynchronous. In the synchronous mode all components are updated before starting a new update cycle. In the asynchronous mode no restriction is placed on the order of updates. Instead they are made as quickly as possible.

To provide a hardware abstraction layer *robot servers* are used to provide access to various physical devices, such as sensors and actuators. ADE also provides several tools and utilities which makes it a complete agent development environment, such as graphical user interfaces and runtime inspection and monitoring support.

One difference compared to DyKnow is the lack of explicit support for time.

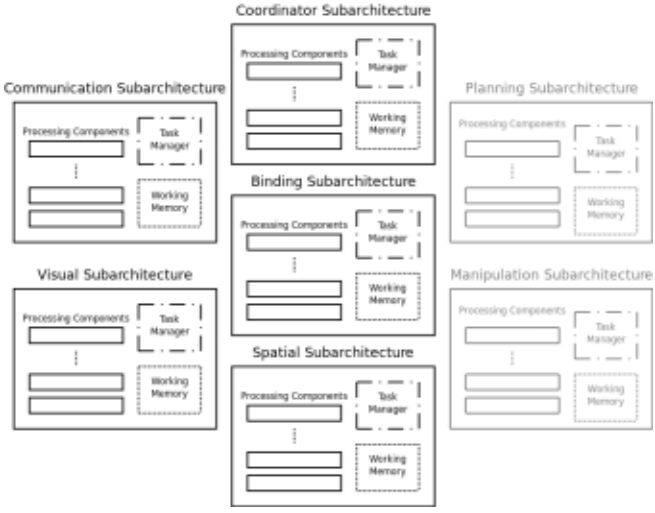


Figure 11.3: Some example subarchitectures (from Hawes, Zillich, and Wyatt (2007)).

Even though data can be time-stamped this information is not taken into consideration by the toolkit itself. Another difference is that ADE lacks the possibility to declaratively specify what data a component is interested in.

11.4.2 CAST/BALT

The CoSy Architecture Schema Toolkit (CAST) is developed in order to study different instantiations of the CoSy Architecture Schema (Hawes et al., 2007; Hawes, Wyatt, and Sloman, 2006; Hawes, Zillich, and Wyatt, 2007). 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.

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

An architecture instantiation of the CoSy Architecture Schema (CAS) consists of a collection of interconnected subarchitectures (SAs) (Figure 11.3). Each subarchitecture contains a set of processing components that can be connected to sensors or actuators, and a working memory which acts like a blackboard within the subarchitecture (Figure 11.4 on the next page).

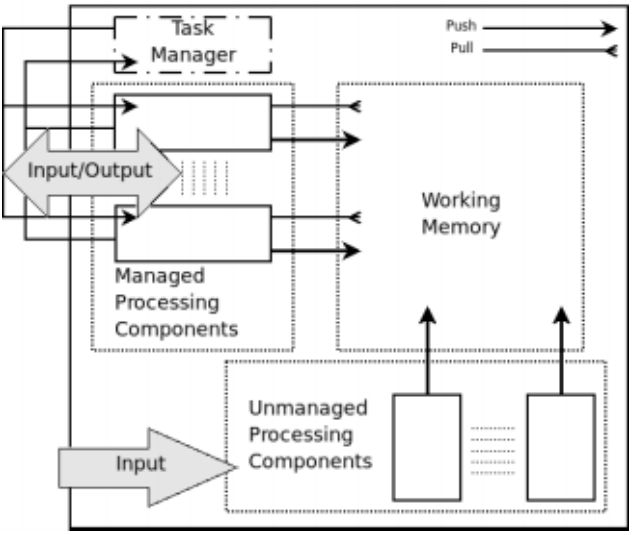


Figure 11.4: The CAS Subarchitecture Design Schema (from Hawes, Zillich, and Wyatt (2007)).

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. The general principle is that processing components work concurrently to build up shared representations. The SAs work concurrently on different sub-tasks, and the components of a SA work on different parts of a sub-task.

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. Change objects generated as a result of a change to working memory are the primary mechanism for distributing information through the architecture.

CAST is implemented using CORBA and has been used for several different robots in the CoSy EU project. Like many of the other frameworks, CAST does not have any explicit support for time. The support for specifying the information a component is interested in is also limited to change objects.

11.4.3 CLARAty

The Coupled Layer Architecture for Robotic Autonomy (CLARAty) is designed for improving the modularity of robotic system software while coupling the interaction of autonomy and control more tightly (Nesnas et al., 2006; Nesnas, 2007;

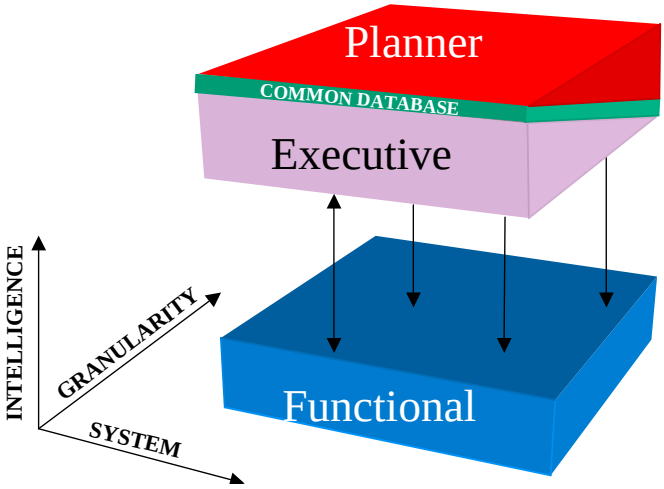


Figure 11.5: An overview of CLARAty (from Volpe et al. (2001)).

Volpe et al., 2001). The architecture consists of two layers, the *Decision Layer* and the *Functional Layer* (Figure 11.5). The Decision Layer provides the high-level autonomy of the system. It reasons about global resources and missions constraints. The Functional Layer provides abstractions of the system and adapts the abstract components to real or simulated devices. Unlike conventional three-level architectures CLARAty makes a separation between *levels of granularity* and *levels of intelligence*. While the Decision Layer has a higher level of intelligence both the Decision Layer and the Functional Layer cover the whole range of different levels of granularity.

The Functional Layer consists of an object-oriented hierarchy of classes which provide basic functionality of system operation, resource prediction, state estimation, and status reporting. Two fundamental notions of CLARAty are: Abstractions at various levels of granularity and encapsulation of information at the appropriate levels of the hierarchy. The Functional Layer provides three main types of abstractions: *Data structures*, *physical components* providing abstractions of physical objects, and *functional components* encapsulating algorithms. The functional and physical components provide interface definitions and implementations of basic functionality, manage local resources, and support state and resource queries by the Decision Layer.

The Decision Layer is a global engine that reasons about system resources and mission constraints. It includes planners, executives, schedulers, activity databases, and rover and planner specific heuristics. This layer plans, schedules, and executes plans. It also monitors the execution and modifies the plans if necessary. In the current instantiation they have a tight coupling between the planner CASPER (Estlin et al., 2000) and the executive TDL (Simmons and Apfelbaum, 1998).

CLARAty supports both push and pull models of data flow. It provides generic

interfaces for bridging the timing requirements of consumers and the actual data flow provided by specific devices. A consumer can choose whether to force a new update, access the most recently stored data, or retrieve a *data source object*. A consumer can customize the timing constraints of a data source object. When new information becomes available all consumers who have registered an interest in the data source will be notified. If new data is not available within the timing constraints then the consumer will be notified about the violation and has the option of forcing the data source to update itself. This feature makes CLARAty one of the few frameworks together with DyKnow that supports the explicit specification of timing constraints.

11.4.4 CoolBOT

CoolBOT is a component-based programming framework for developing robotic systems without imposing any specific architecture (Cabrera-Gámez, Domínguez-Brito, and Hernández-Sosa, 2001; Domínguez-Brito et al., 2007; Fernádez-Pérez et al., 2004). An application consists of a network of asynchronously interacting *components*. Each component is modeled as a *Port Automaton* where all the communication is through input and output *ports* (Steenstrup, Arbib, and Manes, 1983). Transitions in the port automata of a component are triggered by events caused either by incoming data, internal conditions, or a combination of these.

A *connection* between two components consists of an input/output port pair. Data is transmitted through a connection in discrete units called *port packets*. There are two types of port packets, *event packets* that signal events and *data packets* which contains data. Each port can only transmit one type of packet and the two ports in a connection must be of the same type.

There are five types of connections depending on the kind of ports connected. A *poster* output port provides a finite circular buffer of data packets that are made available to poster input ports. This is similar to posters provided by GenoM as described later. A *tick* output port emits event packets to connected tick input ports. This can be used to associated timers with components. A *generic* output port can be connected to either a *LIFO*, *FIFO*, or *unbounded FIFO* input port. In this connection, the input port instead of the output port buffers the packets. This allows a consumer to process data at its own pace disregarding the rate with which data is produced.

The communication through a connection can follow either a push model or a pull model. In the push model the producer pushes the data to the connected consumers while in the pull model a consumer requests the data from a producer. Pull connections can only be created from poster, FIFO, and unbounded FIFO connections.

Another feature of CoolBOT is that it puts a special focus on the *robustness* of its components. A component is considered robust if it is able to monitor its own performance, adapt to changing operating conditions, and recover from errors that it can detect. If a component detects an error which it can not recover from then it should announce the error through its interface and then wait in an idle state until

the error can be handled by some external component.

CoolBOT provides both a formal framework for describing components as port automata and a wide variety of different types of ports. There is however no explicit support for time, synchronization, or for specifying sampling rates.

11.4.5 GenoM

GenoM provides a language for specifying functional modules and automatically generate implementations of these modules according to a generic model (Fleury, Herrb, and Chatila, 1997; Mallet, Fleury, and Bruyninckx, 2002). It was developed in order to specify and integrate functional modules in a distributed hybrid robotic architecture. GenoM also generates an interactive test program and interface libraries to control the module and to read its output. A *module* is a software entity that offers *services* related to a physical (sensor, actuator) or a logical (data) *resource*. This makes the module responsible for the resource. It should control the resource including detecting failures and recovering from these to ensure the integrity of the resource.

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 will include 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 identifier of the poster. This allows services to access the value of any poster in their own pace. Since a service might be executed periodically it is possible to poll a poster with a certain sample period.

GenoM provides a very mature and robust module specification language which has been used in several projects (Ingrand et al., 2007; Mallet et al., 2007). Recently work has been made to formally verify applications developed using GenoM (Basu et al., 2008). From a knowledge processing perspective, it provides support for both synchronous and asynchronous polling of data but not for asynchronous notification of the availability of new data. Neither does GenoM provide support for synchronization or the specification of the data required by a service.

11.4.6 MARIE

The Mobile and Autonomous Robotics Integration Environment (MARIE) is a middleware framework for developing and integrating new and existing software for robotic systems (Côté et al., 2004, 2005, 2006; Côté, Champagne, and Michaud, 2007; Côté, Létourneau, and Michaud, 2007). It is designed according to three software requirements:

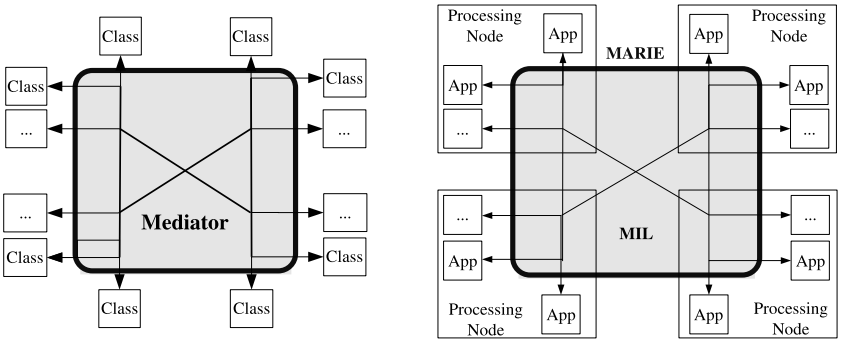


Figure 11.6: Original Mediator Pattern (left) and MARIE's Distributed Mediator Adaptation (right) (from Côté, Létourneau, and Michaud (2007)).

1. Reuse available solutions. To avoid having to start from scratch and reinvent the wheel it is necessary to be able to integrate existing software components even though they have been developed independently with different design requirements.

2. Support multiple sets of concepts and abstractions. This is necessary to support different types of components and applications from high-level decision making, to processing of sensor data, to motor control which have their own objectives and requirements.

3. Support a wide range of communication protocols, communication mechanisms, and robotics standards. Since there is no standard protocol and the robotics community is still exploring different approaches it is important to be able to integrate and adapt existing communication protocols.

To implement distributed applications using heterogeneous softwares, MARIE uses a Mediator Interoperability Layer (MIL) adapted from the Mediator design pattern (Gamma et al., 1994). The Mediator design pattern consists of a centralized control unit (called Mediator) which interacts with a number of classes (called Colleagues) independently to coordinate the global interactions among the classes. In MARIE, the MIL acts as the Mediator among any number of components which are the Colleagues (Figure 11.6). The use of a mediator promotes a loose coupling between components by replacing a many-to-many interaction model with a one-to-many interaction model. Each component can use its own communication protocols and mechanisms as long as the MIL supports it. This provides a way to exploit the diversity of communication protocols and mechanisms.

The development of robotic applications using MARIE is based on reusable software blocks, referred to as components, which implement functionalities by encapsulating existing applications, programming environments, or dedicated algorithms. Components are configured and interconnected to implement the desired system, using the software applications and tools available through MARIE.

To allow connections between components they can have ports. Two ports that are connected allow the components to communicate. Each component has a director port, a configuration port, and possibly input and output ports. The director port is used to control the execution of the component and the configuration port is used to configure the input and output ports.

Four types of components are used in the MIL:

1. Application Adapter (AA): A component which interfaces to applications within the MIL and enables them to interact with each other through MARIE's port-based communication interface.

2. Communication Adapter (CA): A component that makes communication between different components possible by adapting incompatible communication mechanisms and protocols or by implementing traditional routing communication functions.

3. Application Manager (AM): A system level component that manages, on local or remote processing nodes, Application Adapters and Communication Adapters. The initialization, configuration, starting, stopping, suspending, and resuming of Application and Communication Adapters are handled by an Application Manager.

4. Communication Manager (CM): A system level component that dynamically manages, on local or remote processing nodes, the communications mechanisms (socket, port, shared memory, etc.).

To integrate a set of applications into a working system an adapter for each application must be developed. These adapters can then be connected either using the existing communication mechanisms supported by MARIE or through communication adapters. Examples of communication adapters are 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. An example from Côté, Létourneau, and Michaud (2007) where three applications are integrated is shown in Figure 11.7 on the following page. Interconnections using port communication abstraction are illustrated with a small dot between communication links represented by arrows.

The strength of MARIE to integrate many different approaches was shown by the development of Spartacus, a socially interactive autonomous mobile robot (Côté, Létourneau, and Michaud, 2007; Michaud et al., 2007). Spartacus is equipped with a laser range finder, a pan-tilt-zoom color camera, eight microphones, and a touch screen. The processing is done by two onboard computers. The application developed for the 2005 AAAI Mobile Robot Challenge integrated autonomous navigation, vision processing, sound processing, and communication through the touch screen display. An overview of some of the components are shown in Figure 11.8 on page 221.

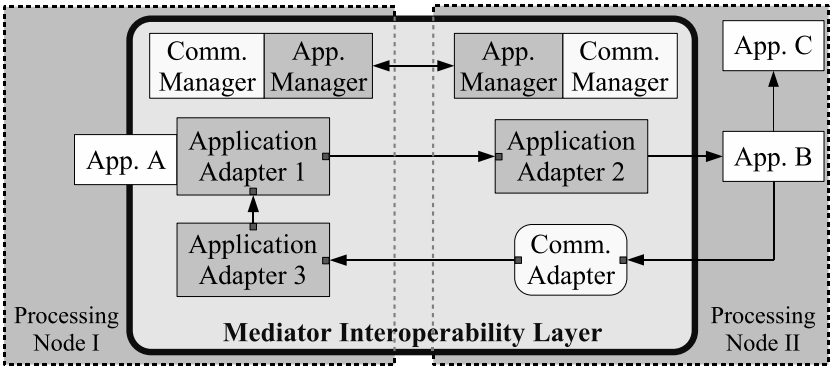


Figure 11.7: MARIE's Component Framework Using the Mediator Interoperability Layer (from Côté, Létourneau, and Michaud (2007)).

Spartacus shows that MARIE has the necessary functionality to integrate many different applications. To our knowledge MARIE does not provide any support for a component to specify the desired properties of its input which DyKnow supports through its policies. Neither does MARIE 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.

11.4.7 Miro

Miro is an object-oriented middleware for mobile robot applications (Kraetzschmar et al., 2002; Krüger et al., 2006; Utz et al., 2002). It is implemented using the TAO/ACE CORBA framework (Object Computing, Inc., 2003). Miro consists of three layers interwoven with two CORBA layers (Figure 11.9 on page 222):

1. The *Miro Device Layer* provides platform dependent object-oriented interface abstractions for sensor and actuator facilities.

2. The *Miro Service Layer* provides active service abstractions for sensors and actuators via CORBA IDL descriptions and implements these services as network transparent and platform independent objects. Event-based communication is provided by the CORBA notification service.

3. The *Miro Class Framework* provides a number of common functional modules for mobile robot control, such as mapping, self localization, behavior generation, path planning, logging, and visualization.

Each sensor and actuator is modeled by an object in the Service Layer which can be controlled and queried through methods. A robot can therefore be viewed as an aggregation of sensor, actuator, and cognitive objects which can trade information and services in an agent-like manner.

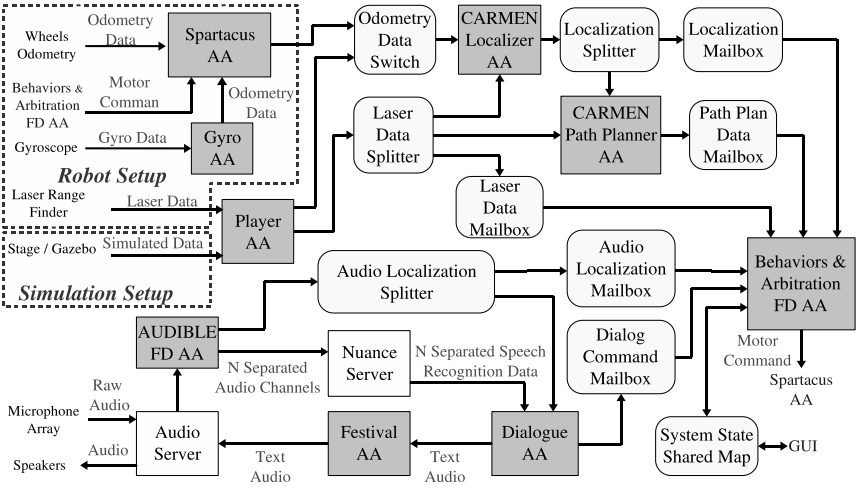


Figure 11.8: An overview of some of the components integrated in Spartacus using MARIE (from Côté, Létourneau, and Michaud (2007)).

Miro has been used on three robot platforms, the B21, the Pioneer-1, and the Sparrow-99 robots which are equipped with different sensors, actuators, and computational resources. The robots have been used in different scenarios including office delivery and robotic soccer. Miro has also been used to implement higher-level functionalities such as Monte-Carlo localization methods, hybrid multi-representation world modeling system, and a hierarchical system of navigation planners.

11.4.8 Orca

Orca is an open source project developing a component-based robotics framework (Brooks et al., 2005, 2007; Kaupp et al., 2007; Makarenko, Brooks, and Kaupp, 2006, 2007). The main focus of the project is to develop a framework that imposes minimal design constraints. An Orca *application* consists of a set of *components* which run asynchronously, communicating with each other using well-defined *interfaces*. Each component provides a set of interfaces and requires another set of interfaces. Orca provides the framework for defining and implementing these interfaces so that they are interoperable and therefore reusable.

To distribute components and to provide platform and language transparency Orca is implemented on top of the Internet Communication Engine (Ice) which is an object-oriented middleware framework similar to CORBA but more modern (Henning, 2004). Each interface is defined using the Slice interface definition language provided by Ice. To allow components to communicate using a publish/subscribe model Orca uses the IceStorm event service provided by Ice, which is similar to the event and notification services provided by CORBA.

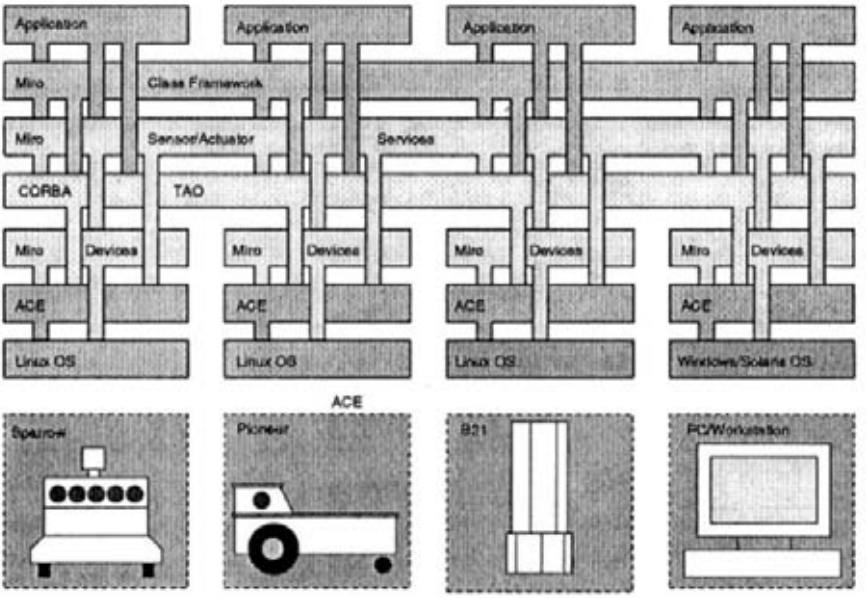


Figure 11.9: An overview of the layers of Miro (from Utz et al. (2002)).

To summarize, Orca provides a component-based framework implemented as a relatively thin layer on top of the Ice middleware. No special support for knowledge processing besides publish/subscribe communication is provided.

11.4.9 Orocos

The aim of the Orocos (Open Robot Control Software) project is to develop a general-purpose and modular framework for robot and machine control and make it available as free software (Bruyninckx, 2001, 2008). It consists of 4 C++ libraries: The Real-Time Toolkit, the Kinematics and Dynamics Library, the Bayesian Filtering Library, and the Orocos Component Library. Of these libraries it is the first that is relevant for this thesis since it provides the real-time infrastructure and the functionalities to build interactive and component-based robotic applications.

An Orocos *application* consists of distributed software *components* which are built and connected using the Real-Time Toolkit. A component encapsulates a *process* of arbitrary complexity and can be interfaced through properties, events, commands, methods, and data flow ports (Figure 11.10). *Properties* are parameters of the component which can be used to configure it. *Events* allow the component to react to changes in the application using a finite state machine. *Commands* are given to a component to command it to reach a goal such as move to a certain position. Since the execution of a command is not expected to finish instantaneously a command request is non-blocking. *Methods* are used to interface computations which return an immediate result. Finally, *data flow ports* implement the Port-

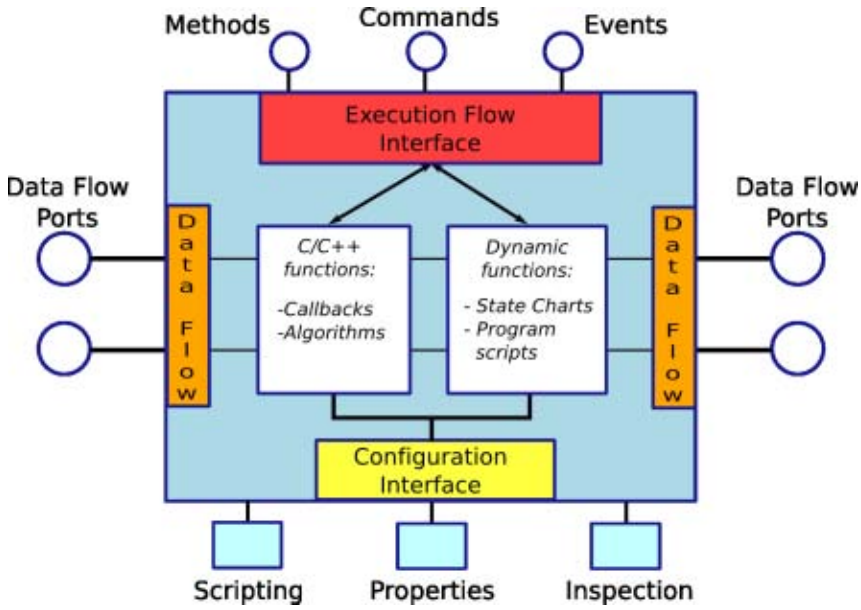


Figure 11.10: An overview of an Orocos Component (from Soetens (2007)).

Connector design pattern and provide a push-based data transport mechanism to send buffered or unbuffered data between components. If a port has a buffer then it is possible to query the port for historic data.

The components provided by Orocos are very general and ought to be appropriate for almost any imaginable process. The data flow ports provide a publish/subscribe communication model between components while the methods and properties support on-demand communication. Since Orocos has periodic tasks it is possible to create a component which generates data with a specific sampling period. However, besides the property interface no support is provide for specifying the input and output of a component. Neither is there any explicit support for synchronization, even though Orocos provides a synchronized clock service.

11.4.10 Player/Stage

Player/Stage (Collett, MacDonald, and Gerkey, 2005; Gerkey, Vaughan, and Howard, 2003; Vaughan and Gerkey, 2007) is a software tool for developing multi-robot, distributed robot, and sensor network systems. Player is a robot device server that provides network transparent robot control. It seeks to minimize the constraints on controller design as it is device independent, non-locking, and language-neutral. Stage is a lightweight, highly configurable robot simulator which supports large populations.

The design of Player is heavily influenced by the design of the UNIX operat-

ing system. For example, one of the main principles is to hide the details of the underlying hardware that varies from robot to robot. This is done through the use of a *device model* where a robot functionality is seen as a *device* which is controlled through an *interface*. An implementation of a device is called a *driver*. By programming against a device interface any robot which has a driver for that device can be used. This provides a clear abstraction to well-defined devices such as position estimation, laser scanners, and sonars.

A driver can either send data directly to another driver or a client or it can send data to all subscribed clients. Incoming data are placed on a *message queue*. It is up to each driver or client to process the messages on the queue. The delivery of data is handled by a separate transport layer. This layer is usually implemented using TCP sockets but other transport protocols such as CORBA and JINI could be used as well. A program that contains a set of drivers is called a *Player Server*.

The Player/Stage project is mature, but mainly provides a hardware abstraction layer hiding the hardware differences between different robots. The communication infrastructure provides basic support for publish/subscribe communication between clients through a Player server. There is no support for specifying the properties of the input or output of a client.

11.4.11 ROCI

ROCI (Remote Objects Control Interface) is a self-describing, object-oriented, strongly typed programming framework for developing applications involving multi-robot teams (Chaimowicz et al., 2003; Cowley, Chaimowicz, and Taylor, 2006; Cowley, Hsu, and Taylor, 2004). An application in ROCI consists of a number of distributed *modules* that have typed input and output *pins*. A module encapsulates a *process* which acts on the data from its input pins and makes results available to its output pins. The pins of the different modules can be connected in a variety of ways to create different application instances. Only pins of the same type can be connected. The connections can also be changed dynamically at run-time. A collection of modules that work together to accomplish some result can be organized into a *task*.

When a module assigns data to an output pin all the connected input pins will be *notified* about the availability of new data. They can then decide whether they want to get this data or not. Since data are time-stamped a module can compare the age of its current input value with the freshly available one. When all consumers have processed the notification the output data is removed.

ROCI supports easy and efficient distribution of modules over many nodes where modules can communicate using a typed push-based mechanism. The information processing infrastructure does not support synchronization or the specification of properties of data streams apart from their data type.

11.4.12 S* Software Framework

The S* Software Framework (Rotenstein et al., 2007; Tsotsos, 1997) supports the development of intelligent control systems in the behavior-based paradigm. It provides a set of design principles for integrating representations in a behavior-based controller, which are necessary for learning, cognition, and perception. This is done by extending a behavior-based architecture with *representations* which are used by behaviors both to access information extracted through perception and reasoning, and to send commands to actuators.

A *behavior* maintains two sets of references to representations, one used for input called *event representations* and the other used for output called *action representations*. This allows a behavior to be implemented by a sense–model–plan–act process where it first senses a subset of its event representations, constructs an internal model that is used to reason about the inputs and produce a plan, which is then carried out by updating the action representations of the behavior. Behaviors are decoupled by their use of shared representations.

To determine when a behavior is to be executed it maintains a *window of attention* by monitoring a subset of its event representations for relevant state changes. If such a change, called an *event*, is detected the behavior is triggered as soon as it becomes idle. Each event has a *trigger condition* which is a first-order predicate evaluated each time an operation is performed on one of the event representations.

The S* Software Framework is implemented as a class hierarchy in C++, where Behavior and Representation are two abstract base classes which can be extended to implement behaviors and representations. A representation is a shared data structure with state. Since representations are shared it is important that the internal data structures are locked, which may cause calls to block if some other part is updating it. Representations can be chained together to describe an incremental transformation of sensor data. However, no support is provided for explicit time or for distributing the computations over multiple computers.

11.4.13 SPQR-RDK

SPQR-RDK is a modular robot development toolkit for mobile robots developed by Università di Roma "La Sapienza" (Farinelli, Grisetti, and Iocchi, 2005, 2006). It provides an infrastructure for executing tasks and sharing information, a remote inspection capability, and a common robot hardware interface. It has been used to develop robots for RoboCup soccer, RoboCup rescue, and RoboCare.

The Robot Hardware Interface module encapsulates the functionalities of the underlying robot platform by providing abstractions for kinematics and devices such as sensors and actuators. The main concepts are *Robot* and *Device* controlling the robot kinematics and the devices respectively. Each device is connected to a robot. A specific implementation of a device or a robot is called a *driver*. Both devices and robot drivers can be replaced by simulators or logfile players. The hardware configuration is described in a configuration file.

The Remote Inspection Server provides a general mechanism for remotely inspecting the internal status of each component of an application and dynamically

choose what to monitor and when, while limiting the network bandwidth used and the computational overhead. The remote introspection capability provided by SPQR-RDK uses TCP for reliable communication, a separate thread to reduce interference with other processes, and a publish/subscribe communication model to disseminate information.

The Task Manager allows a user to dynamically load modules, to specify their execution features (execution period, scheduling policy, priority, and so on), and to export the information to be shared among them. There are three types of tasks: *Asynchronous tasks* corresponding to normal threads, *periodic tasks* corresponding to asynchronous tasks that are re-spawned with a fixed period, and *serial tasks* grouping tasks together whose execution are serialized within the group.

Since using shared memory without any data access policy is difficult the management of shared data can become very complex. By using conventions, either explicitly or implicitly, the modularity is reduced. If a module needs information provided by some other module, then it needs to know *where* to read the information and *when* the information is available. By providing a specification in the form of the *type* of information provided by a module, the coupling between modules is reduced.

Therefore the task manager has a *Shared Information Register* (SIR) sub-component that provides a possibility to exchange information through a shared memory specified by their types. A producer publishes information in the SIR under a name and consumers can read the information from the SIR at their own pace. No notification mechanism for informing consumers when new information seems to be available and no information has been found on what can be specified in the type. To manage distribution aspects they have a *pass-through* mechanism which allows data to be transferred between computers.

11.4.14 YARP

Yet Another Robot Platform (YARP) is an open-source project that provides a platform for developing applications that are real-time, computation-intensive, and involve interfacing with diverse and changing hardware (Fitzpatrick, Metta, and Natale, 2008; Metta, Fitzpatrick, and Natale, 2006). It is developed using the Adaptive Communication Environment (ACE) (Huston, Johnson, and Syyid, 2003) to provide support for different operating systems including Windows, Linux, Mac OSX, and QNX. YARP provides a communication abstraction, a device abstraction, and a library of signal processing routines for audio and image processing. It has been used in many applications on a number of different robots (Beltran-Gonzalez and Sandini, 2005; Brooks et al., 1999; Edsinger-Gonzales and Weber, 2004).

Communication is supported in the form of *port* objects. A port is an active object managing multiple connections for a given unit of data either as input or output. An input port can receive data from multiple connections at different rates and using different protocols. An output port can send data to many destinations, where each destination may read the data at a different rate. The use of several different protocols allows YARP to exploit their different characteristics. Each

port is typed and has a unique name which is registered in a *name server*.

When data is written to an output port it is only sent to those connected input ports which are not busy processing the previous input. In order for all connected input ports to get a new value the output port has to wait for them to finish. This can be done in two different ways. Either the output port waits before writing the new data (called *wait-before*) or it waits after it has written data and thereby is certain that the input ports are available when a new value is produced (called *wait-after*). The default strategy is not to wait (called *no-wait*).

While data is being processed by an input port there are three strategies for handling new input. The *single-buffer* strategy does not allow new data to be received until the previous data has been processed. With the *double-buffer* strategy an input port stores two data items. When the first item is being processed a second item can be received. If more than one item arrives while the first one is processed then only the latest is kept. This strategy minimizes the data latency. With the *triple-buffer* strategy three data items are stored and there will always be a new item ready to be processed if the producer process is at least as fast as the client process. As with the double-buffer strategy only the latest data item is kept. This strategy maximizes throughput.

As with many of the other frameworks YARP provides a light-weight and highly functional communication infrastructure on a quite low level. No support is provided for representing time, synchronizing data, or specifying the desired properties of inputs and outputs.

11.4.15 Discussion

Even though there exist many different frameworks on different abstraction levels, the support provided for knowledge processing is usually limited to distributed components which can communicate. Most frameworks provide publish/subscribe communication while some also support the combination of push and pull communication where a producer pushes information to a consumer that buffers the data to be consumed on demand. What all these frameworks lack is ways for consumers to specify the information that they 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 S* and 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 the valid time and the 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 also support the 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 which 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.

11.5 Summary

This chapter has compared stream-based knowledge processing middleware to distributed real-time databases, control architectures, and robotics frameworks providing general support for integrating sensing and reasoning. The main conclusion with respect to distributed real-time databases is that they provide complementary solutions that can be used to further develop knowledge processing middleware and provide real-time guarantees. Control architectures currently use partial and specialized solutions to bridge the gap between sensing and reasoning. This accentuates the need for DyKnow, since there are many disparate partial solutions which would benefit from being integrated. Regarding existing robotics frameworks the main conclusion is that the support provided for knowledge processing is usually limited to distributed components which can communicate. These frameworks generally lack explicit representation of time and ways for consumers to specify the information they would like to have. These two concepts are central to knowledge processing middleware such as DyKnow.

Chapter 12

Conclusions

12.1 Summary

As robotic systems become more and more advanced the need to integrate existing deliberative functionalities such as chronicle recognition, motion planning, task planning, and execution monitoring increases. To integrate such functionalities into a coherent system it is necessary to reconcile the different formalisms used by the functionalities to represent information and knowledge about the world. To construct and integrate these representations and maintain a correlation between them and the environment it is necessary to extract information and knowledge from data collected by sensors. However, deliberative functionalities tend to assume symbolic and crisp knowledge about the current state of the world while the information extracted from sensors often is noisy and incomplete quantitative data on a much lower level of abstraction. There is a wide gap between the information about the world normally acquired through sensing and the information that is assumed to be available for reasoning about the world.

As physical autonomous systems grow in scope and complexity, bridging the gap in an ad-hoc manner becomes impractical and inefficient. Instead a principled and systematic approach to closing the sense-reasoning gap is needed. At the same time, a systematic solution has to be sufficiently flexible to accommodate a wide range of components with highly varying demands. We therefore introduced the concept of *knowledge processing middleware* for a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent.

To describe the desired features of knowledge processing middleware a set of design requirements were specified. Knowledge processing middleware should:

- Support integration of information from distributed sources (requirement 1a),
- support processing on many levels of abstraction (requirement 1b),
- support integration of existing reasoning functionalities (requirement 1c),

- support quantitative and qualitative processing (requirement 2),
- support bottom-up data processing and top-down model-based processing (requirement 3),
- support management of uncertainty (requirement 4),
- support flexible configuration and reconfiguration (requirement 5), and
- provide a declarative specification of the information being generated and the information processing functionalities available (requirement 6).

Then, *stream-based* knowledge processing middleware was proposed as one specific type of middleware which provides an appropriate basis for satisfying the requirements. Due to the need for incremental refinement of information at different levels of abstraction, we model computations and processes within the stream-based knowledge processing framework as active and sustained *knowledge processes* working on and producing *streams*. This step provides a considerable amount of structure for the integration of the necessary functionalities, but still leaves certain decisions open in order to avoid unnecessarily limiting the class of systems to which stream-based knowledge processing middleware is applicable.

We then presented a knowledge processing middleware framework called DyKnow which supports the integration of diverse data and knowledge sources with different existing knowledge processing systems in such a way that the result is useful for a cognitive robotic system deployed in an uncontrolled environment. DyKnow supports generating partial and context dependent stream-based representations of past, current, and potential future states at many levels of abstraction in a timely manner. Contextual generation of world state is essential in distributed contexts where contingencies continually arise which often restrict the amount of time a system has for assessing situations and making timely decisions. It is our belief that autonomous systems will have to have the capability to determine where to access data, how much data should be accessed, and at what levels of abstraction it should be modeled. We have provided initial evidence that such a system can be designed and deployed and described several scenarios where such functionality is useful.

To show the versatility and utility of DyKnow two symbolic reasoning engines were integrated into DyKnow. By integrating these reasoning engines into DyKnow, they can be used by any knowledge processing application. Each integration therefore extends the capability of DyKnow and increases its applicability.

The first reasoning engine is a metric temporal logical progression engine. Its integration is made possible by extending DyKnow with a state generation mechanism to generate state sequences over which temporal logical formulas can be progressed. A task planner, TALplanner, is then extended with support for annotating plans with monitor formulas in an approach to integrating planning and execution monitoring. These formulas are evaluated using the progression engine as the plan is executed. This functionality has been used to implement a logistics scenario as part of an emergency service application.

The second reasoning engine is a chronicle recognition engine for recognizing complex events such as traffic situations. The integration is facilitated by extending DyKnow with support for anchoring symbolic object identifiers to sensor data in order to collect information about a physical object using the available sensors. The resulting object linkage structures not only anchor symbols to sensor data, they also perform incremental object classification to make the classification more and more specific. Information about the objects in the environment can be collected using object linkage structures and provided as input to the chronicle recognition engine, which can then recognize scenarios or behaviors involving these objects. As a concrete example, a traffic monitoring application was developed which takes streams of color and thermal images, recognizes and tracks cars in these streams, and then detects traffic patterns involving the tracked cars.

The first two applications focus on how DyKnow can support closing the sense-reasoning gap within a single agent. To show that DyKnow also has a potential for multi-agent knowledge processing, an extension was presented which allows agents to federate parts of their local DyKnow instances to share information and knowledge. Using the DyKnow federation concept, a virtual proximity sensor was developed where a set of UAVs share their current positions and use the execution monitoring functionality to detect if two UAVs get too close to each other.

Finally, it was shown how DyKnow provides support for the functionalities on the different levels in the JDL Data Fusion Model, which is the de facto standard functional model for fusion applications. Note that the focus here was not on individual fusion techniques but rather on an infrastructure that permits use of many different fusion techniques in a unified framework.

12.2 Conclusions

In the introduction six requirements on knowledge processing middleware were presented (Section 1.2.1 on page 11). 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.

Requirement 1a: Support integration of information from distributed sources. DyKnow satisfies this requirement by virtue of three features: CORBA-based implementation, explicit support for representing time, and a stream synchronization mechanism.

Since DyKnow is implemented on-top of CORBA, its extensive support for distributed applications is leveraged. For example, it allows different knowledge processes within the same knowledge processing application to be distributed over different processes and machines as long as the computing nodes are connected with some form of network.

All samples in DyKnow contain both a valid time and an available time. This provides explicit support for modeling important aspects of a distributed system

such as when information is available and when information is valid. It also allows DyKnow to model the delay of the information as the difference between the available and the valid time. This gives DyKnow, for example, the possibility to determine which information is the most current, how to control different types of prioritization and scheduling mechanisms to improve the throughput of the system, and whether the delays in a system are too long.

The explicit representation of time can also be used to guarantee that information arrives in the order it was produced, even though different samples might have different delays. If the delay of a sample is too long, or if a sample is lost, then it is possible to approximate the missing value by using the available information. These functionalities are put to good use in the stream synchronization mechanism presented in Section 7.8. A major benefit of this mechanism is that the streams from the different sources in the system do not have to be synchronized when they enter the system, even though some functionalities require synchronized streams of states.

Requirement 1b: Support processing on many levels of abstraction. DyKnow is designed to provide both general and specific support for knowledge processing at multiple levels of abstraction.

General support is provided through fluent streams, where information can be represented at any abstraction level from raw sampled sensor data and upwards and where each element can be unstructured or structured with arbitrary complexity. The use of knowledge processes also provides general support for arbitrary forms of processing. Thus, the information flowing through the system and the processing performed on this information is not arbitrarily limited by the stream-based knowledge processing framework or its implementation.

At the same time, DyKnow is explicitly designed to be extensible and to provide support for information structures and knowledge processing that is more specific than arbitrary streams and yet useful for a wide array of applications. For example, more specific stream support is currently provided for state streams, object state streams, and streams of recognized chronicles. 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, together with a suitable framework for further extensions.

That the support is enough to provide an appropriate framework for supporting all the functional abstraction levels in the JDL Data Fusion Model was argued in Chapter 10.

Requirement 1c: Support integration of existing reasoning functionalities. By providing a general representation in the form of streams, any reasoning functionality whose inputs can be modeled as streams and where the desired properties on the input streams can be described by policies can easily be integrated using DyKnow. Due to this general representation it is enough to describe how to model

the input to a reasoning functionality in the form of streams to integrate it. As two concrete examples, we have shown how progression of temporal logical formulas (Chapter 7) and chronicle recognition (Chapter 8) can be integrated using DyKnow.

Requirement 2: Support quantitative and qualitative processing. First, since fluent streams can contain anything, from real values to images to object structures to qualitative relations, DyKnow provides basic support for both quantitative and qualitative processing. The structured content of samples also allows quantitative and qualitative information to be part of the same sample.

Second, DyKnow has explicit support for combining qualitative and quantitative processing in the form of chronicle recognition, metric temporal logical formulas, and object linkage structures. The objects in an object linkage structure often contain both quantitative and qualitative attributes. For example in the traffic monitoring application, a vision object contains quantitative attributes such as size and position while an on road object contains qualitative information such as which road the object is on and whether it is in a crossing or not. Both chronicles and temporal logical formulas support expressing conditions combining quantitative time and qualitative features and thus support both quantitative and qualitative processing.

Requirement 3: Support bottom-up data processing and top-down model-based processing. Streams are directed, which gives the application programmer the possibility to do both top-down and bottom-up processing. For example, a knowledge process taking a stream of high-level objects can create a stream containing information about how the image processing system should focus its detection and tracking of these objects.

In the DyKnow implementation, it is also possible to change policies for streams at run-time. This means that it is possible for knowledge processes on one level of abstraction to control or influence processing on other abstraction levels by replacing their policies. This is another form of bottom-up and top-down processing.

Object linkage structures are an example where explicit support for bottom-up processing is provided. By looking at the incoming data a more and more refined classification of the data is made. So far, the object linkage structures have been purely bottom-up. However, we believe that they are well suited for combining both bottom-up and top-down processing. For example, if an object has been classified as a car, then this information can be used to reason about how cars can move in order to be able to find and reidentify the same car later.

Chronicle recognition is a typical example of top-down model-based processing. By describing a general pattern, all its instances are detected by the recognition engine. When a chronicle is added the streams required to recognize chronicle instances are created. What streams are created is thus partly dependent on what chronicles are active.

Requirement 4: Support management of uncertainty. In principle, DyKnow supports any approach to representing and managing uncertainty which 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 which represents how probable or certain this particular sample is. This information can then be used by the knowledge processes subscribing to this fluent stream.

Apart from this, DyKnow has explicit support for uncertainty in object identities and in the temporal uncertainty of complex events which can be expressed both in quantitative and qualitative terms as was shown in the chapter about chronicle recognition. The use of a metric temporal logic also provides several ways to express temporal uncertainty, as was discussed in Section 7.9.

Another feature which can be used to support the management of uncertainty is that a fluent stream may contain several samples with the same valid time. This can be used by a system which gradually improves the certainty of an approximation by continually refining the estimation. Each new estimation would have the same valid time but a new available time. This provides an explicit representation of the different estimations and when they are made available.

Requirement 5: Support flexible configuration and reconfiguration. The main support for flexible configuration is provided by the declarative specification language KPL (Chapter 4). It allows an application designer to describe the different processes in a knowledge processes application and how they are connected with streams. Each of these streams is described with a policy which specifies its properties. The knowledge processing application can then be created according to the specification by the DyKnow implementation.

The mediation and configuration processes described in Chapter 3 provide ample support for flexible reconfiguration. They allow for example new streams and processes to be created and old ones to be removed at run-time. Even though there is currently no declarative support for mediation and configuration processes, the DyKnow implementation provides the necessary interfaces to procedurally reconfigure a knowledge processing application at run-time.

Requirement 6: Provide a declarative specification of the information being generated and the information processing functionalities available. This requirement is satisfied since DyKnow provides the formal language KPL for declarative specifications of DyKnow knowledge processing applications, as already described when discussing the fifth requirement. The specification explicitly declares the properties of the streams by policies and how they connect the different knowledge processes.

Main Conclusion

The main conclusion of this thesis is that the DyKnow knowledge processing middleware framework provides appropriate support for bridging the sense-reasoning gap in a physical agent. This conclusion is drawn from the fact that DyKnow has

been used to facilitate the development of two complex UAV applications and that it satisfies all the stated requirements for knowledge processing middleware to a significant degree.

DyKnow provides a foundation for integrating different sensor processing approaches as well as different reasoning approaches in a coherent knowledge processing framework. DyKnow can also easily be extended by integrating different reasoning engines. DyKnow therefore opens up many interesting possibilities for using both traditional knowledge representation techniques and traditional sensor processing techniques to reason about the environment of a physical agent.

12.3 Future Work

Even though DyKnow already satisfies the requirements on knowledge processing middleware, there are many opportunities for improving the support. This section gives a number of concrete examples of how the support provided by DyKnow can be improved in order to even better fulfill the requirements.

Requirement 1a: Support integration of information from distributed sources. There are at least three aspects on integrating information from distributed sources: Integrating the information, anchoring common symbols, and fusing the information.

The first aspect is about getting the distributed information into the integrated system. This requires support for communication with the external sources and the adaptation of their information to something which can be used by the integrated system. DyKnow already provides ample support for this.

The second aspect is about agreeing on common symbols such as object identifiers. For example, in the multi-UAV traffic monitoring scenario, it is crucial that the UAVs are able to agree on the identity of the cars they are sharing information about. Assuming that one of the UAVs has anchored a symbol `car1` to its sensor data, one issue is how the first UAV can communicate enough information to the other UAV in order for it to anchor the same symbol to the same physical car. This is a problem that we call *cooperative anchoring*. To extend DyKnow with support for cooperative anchoring would be very interesting and this is something which we are currently working on.

The third aspect is about fusing information about the same entity from many different sources to make a better estimation than any of the individual sources could do on their own. Currently, DyKnow does not provide much explicit support for this, even though its general functionality can be used to solve information fusion problems. It would be interesting to study if there are some general fusion algorithms which could be provided as standard knowledge processes. One requirement would be that the algorithm is useful in many different applications. Another direction could be to explore the concept of *trust*. For example, each source could be associated with a level of trust, which can then affect the decision on whose information should be used in case of conflicting information.

Requirement 1c: Support integration of existing reasoning functionalities. Currently most processing in DyKnow is done on historic or current information. One very interesting direction for future research is to extend DyKnow with explicit support for prediction. This would make it easier to integrate reasoning functionalities that not only reason about the current state, but also about future states of a system. General support for prediction could for example include special policies for describing properties of streams containing predicted values. Another idea could be to maintain several possible future streams given the development so far of a stream.

It would also be interesting to extend the object linkage structures with explicit support for prediction. The reason is that it is often necessary to predict the current state of an object in order to be able to determine if a newly detected object is in fact the same object as one that has been seen previously. One possible approach would be to replace a world object which does not have a vision object linked to it with a simulated world object. This means that as long as the image processing system updates a vision object, this vision object is used to update the current state of the world object. However, when the vision object is no longer tracked, the world object is replaced by a simulated world object which estimates its current state based on the history of the replaced world object and a predictive model of world objects. The world object could also use information from objects further up in the hierarchy to provide constraints on the prediction. When a new world object is created it would be relatively easy to compare it to the simulated world objects.

Since DyKnow provides support for extending itself by integrating different reasoning engines, this leads to the possibility of having many different approaches to solving one particular problem, for example recognizing complex events. One direction for future research could be to take an expressive language for a particular task and depending on the fragment of the language used give the task to the appropriate reasoning engine. It could also be possible to define special policies that can be used to guide the choice of engine, especially for those context dependent trade-offs that can not be deduced from an event definition itself.

Requirement 2: Support quantitative and qualitative processing. To provide further support for bridging the sense-reasoning gap DyKnow could provide explicit support for transforming numeric data into symbolic representations. For example, to define qualitative spatial relations based on quantitative coordinates. One idea could be to provide a component library with standard knowledge processes which would handle common types of transformations. These processes could for example provide support for handling issues where the transformation is not entirely stable due to uncertainty, like we did in the traffic monitoring scenario (Section 8.7.4). An interesting question is whether it is possible to find a set of common transformations which are useful over a wide range of different applications.

Requirement 3: Support bottom-up data processing and top-down model-based processing. As was mentioned earlier, object linkage structures provide

explicit support for bottom-up data-driven reasoning. It would be very useful to make them support both top-down and bottom-up reasoning. One useful extension would therefore be explicit support for top-down reasoning in object linkage structures. For example, if an object linkage structure has been created which links a vision object to a world object to an on road object, then the high-level attributes computed for the on road object could be used to guide the image processing when tracking the vision object. It would also be very interesting to see if the abstract information higher up in the object linkage structure could be used to guide its reidentification. For example, cars usually try to stay on large roads as long as possible. This behavior could be used when trying to find a lost car in a road system.

Requirement 4: Support management of uncertainty. The obvious extension when it comes to management of uncertainty would be to extend the formal DyKnow framework presented in Chapter 4 with explicit support for some form of uncertainty measure either in the form of probabilities, fuzziness, certainty factors, or something else. One major issue is what uncertainty framework provides the most appropriate support for a wide variety of knowledge processing applications. When a choice has been made the formal framework can be extended. What would be especially interesting would be to extend the policies to explicitly specify the requirements on the certainty of the information. For example, if each sample has a probability then a policy could state that a stream should only contain samples which have a probability of at least 95% or where the total probability for the whole stream is at least 90%.

Another approach to handling uncertainty would be to support multiple hypotheses in the object linkage structures. Currently an object is only linked to at most one object on the next level in the hierarchy. For example, only a single on road object can be linked to a world object. An alternative would be to maintain multiple hypotheses, so a world object could be linked in some way to many on road objects. An interesting question is how to handle the case when there are many world objects which could all be the same on road object, but it is not certain which one it is. In this case the system either has to decide which is the most likely object, or somehow maintain several copies of the on road object, each updated by one of the world objects. As more information is collected, some of the hypotheses can be removed since they are no longer plausible. Hopefully, only a single on road object remains after a while. This has consequences not only for the anchoring but also for subsequent steps in the processing, such as determining the qualitative spatial relations between the on road objects.

Requirement 5: Support flexible configuration and reconfiguration. The most interesting direction for future work when it comes to flexible configuration would be to extend the KPL specification language to include support for mediation and configuration processes.

To handle configuration processes, it is necessary to represent the complete history of a knowledge processing application, not only the current configuration which is done now. A configuration process specification would then describe

how a knowledge processing application is transformed by adding and removing streams and processes.

Supporting mediation processes is likely easier. One approach could be to introduce object identifier variables, such as closest_car to refer to the object identifier of the car which is currently the closest. A knowledge process could then specify that its input stream should be the position of the closest car by using a label such as position[closest_car]. It would then be up to DyKnow to replace the input stream as the value of the variable changes. This could also be extended by introducing object identifier set variables, which refer not to a single object identifier but to a set of object identifiers. For example the variable Car could refer to a set of object identifiers which are classified as cars. A knowledge process could then take the position of all cars as input by using a label such as position[Cars]. DyKnow would then make sure that the knowledge process had one input stream for each of the object identifiers in the set.

Another interesting direction would be to explore explicit support for focus of attention in order to automatically and dynamically adjust the policies to maintain a certain global behavior. For example, if a policy states that the samples in a stream should not be delayed with more than 100ms and the current load on the system is so high that this is not possible, a solution could be to reduce the sample period either in this policy or in another. By adjusting the policies of the streams in the system the global behavior of the system can be optimized. One simple approach would be to attach priorities to policies. DyKnow could then first satisfy the high-priority policies and then, if there are still resources left, the rest of the policies. Knowledge processes using low-priority policies would then have to accept that the policies can be changed, by for example increasing the delay or decreasing the sample period.

Requirement 6: Provide a declarative specification of the information being generated and the information processing functionalities available. A very interesting direction for future work would be to investigate how an agent can derive its own policies depending on its needs instead of relying on an application programmer. For example, if an agent could estimate how much resources a particular KPL specification would require, then it could reason about the trade-offs between using different specifications given a fixed amount of resources.

Another interesting direction would be to extend KPL with a concept of quality of information. This can be seen as a generalization of adding an explicit representation of the uncertainty of the information. To make it interesting the policies would have to be extended to specify the desired quality of information. The DyKnow implementation could then use the constraints introduced by the policies when it generates the streams. This could provide another approach to controlling the total computations of a system as was discussed when talking about focus of attention. If the current quality of the information is higher than the required quality, then the system has some slack that it can use to redirect resources to other computations.

12.4 Final Words

This thesis has presented the stream-based knowledge processing middleware framework DyKnow. The presented version of DyKnow should be considered as the current state of an active and constantly evolving development process. We believe that DyKnow, as it is, provides both a powerful tool for developing knowledge processing applications and a good basis for interesting further developments in the area of integrating sensing and reasoning in autonomous systems.

Part V

Bibliography

Bibliography

Abadi, D. J.; Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Stonebraker, M.; Tatbul, N.; and Zdonik, S. B. 2003. Aurora: A new model and architecture for data stream management. *VLDB Journal* 12(2):120–139.

Aguilar, J.; Bousson, K.; Dousson, C.; Ghallab, M.; Guasch, A.; Milne, R.; Nicol, C.; Quevedo, J.; and Travé-Massuyès, L. 1994. TIGER: Real-time situation assessment of dynamic systems. *Intelligent Systems Engineering* 103–124.

Albus, J. 1981. *Brain, Behavior, and Robotics*. McGraw-Hill.

Albus, J. S. 2002. 4D/RCS: A reference model architecture for intelligent unmanned ground vehicles. In *Proceedings of SPIE Aerosense Conference*.

Alur, R., and Henzinger, T. A. 1992. Back to the future: Towards a theory of timed regular languages. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science (FOCS-1992)*, 177–186. IEEE Computer Society Press.

Alur, R.; Feder, T.; and Henzinger, T. A. 1991. The benefits of relaxing punctuality. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-1991)*, 139–152. ACM Press.

Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution and monitoring. In *Proceedings of the 7th National Conference of Artificial Intelligence (AAAI-1988)*, 83–88. AAAI Press / The MIT Press.

Amirijoo, M. 2007. *QoS Control of Real-Time Data Services under Uncertain Workload*. Ph.D. Dissertation, Linköpings universitet. Linköping Studies in Science and Technology, Dissertation No 1143.

Andronache, V., and Scheutz, M. 2003. APOC - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium*, 18–25. AAAI Press.

Andronache, V., and Scheutz, M. 2004. Integrating theory and practice: The agent architecture framework APOC and its development environment ADE. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, 1014–1021. IEEE Computer Society.

Andronache, V., and Scheutz, M. 2006. ADE - an architecture development environment for virtual and robotic agents. *International Journal on Artificial Intelligence Tools* 15(2):251–286.

Arkin, R. C. 1998. *Behavior-Based Robotics*. MIT Press.

Atkin, M. S.; King, G. W.; Westbrook, D. L.; Heeringa, B.; and Cohen, P. R. 2001. Hierarchical agent control: A framework for defining agent behavior. In *Proceedings of Agents 2001*, 425–432.

Atkin, M. S.; Westbrook, D. L.; and Cohen, P. R. 2001. HAC: A unified view of reactive and deliberative activity. In Markus Hannebauer, J. W., and Pagello, E., eds., *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, volume 2103 of *LNAI*. Berlin Heidelberg: Springer-Verlag. 92–107.

Babcock, B.; Babu, S.; Datar, M.; Motwani, R.; and Widom, J. 2002. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*.

Bacchus, F., and Kabanza, F. 1996. Planning for temporally extended goals. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-1996)*, 1215–1222. AAAI Press / The MIT Press.

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence* 22:5–27.

Banavar, G.; Chandra, T. D.; Mukherjee, B.; Nagarajarao, J.; Strom, R. E.; and Sturman, D. C. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS 1999)*, 262–272.

Barbera, A.; Horst, J.; Schlenoff, C.; Wallace, E.; and Aha, D. W. 2003. Developing world model data specifications as metrics for sensory processing for on-road driving tasks. In *Proceedings of the 2003 Performance Metrics for Autonomous Systems (PerMIS) workshop*.

Barringer, H.; Goldberg, A.; Havelund, K.; and Sen, K. 2004. Program monitoring with LTL in EAGLE. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*.

Barringer, H.; Rydeheard, D.; and Havelund, K. 2008. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation*.

Basu, A.; Gallien, M.; Lesire, C.; Nguyen, T.-H.; Bensalem, S.; Ingrand, F.; and Sifakis, J. 2008. Incremental component-based construction and verification of a robotic system. In *Proceedings of the 18th European Conference on Artificial Intelligence*.

Bellifemine, F.; Caire, G.; and Greenwood, D. 2007. *Developing Multi-Agent Systems with JADE*. Wiley.

Beltran-Gonzalez, C., and Sandini, G. 2005. Visual attention priming based on crossmodal expectations. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, 1060–1065.

Ben Lamine, K., and Kabanza, F. 2002. Reasoning about robot actions: A model checking approach. In *Revised Papers from the International Seminar on Advances in Plan-Based Control of Robotic Agents,*, 123–139. Springer-Verlag.

Bibas, S.; Cordier, M.-O.; Dague, P.; Dousson, C.; Lévy, F.; and Rozé, L. 1996. Alarm driven supervision for telecommunication networks: I – Off-line scenario generation and II – On-line chronicle recognition. *Annals of Telecommuncations* 493–508.

Biggs, G., and Macdonald, B. 2003. A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation*.

Bjäreland, M. 2001. *Model-based Execution Monitoring*. Ph.D. Dissertation, Linköpings universitet. Linköping Studies in Science and Technology, Dissertation No 688.

Blasch, E., and Plano, S. 2003. Level 5: User refinement to aid the fusion process. In Dasarathy, B., ed., *Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications*.

Bonarini, A.; Matteucci, M.; and Restelli, M. 2001. Anchoring: Do we need new solutions to an old problem or do we have old solutions for a new problem? In *Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems: Papers from the 2001 AAAI Fall Symposium*, 79–86. Menlo Park, CA: AAAI Press.

Bonasso, P. R.; Firby, J. R.; Gat, E.; Kortenkamp, D.; Miller, D. P.; and Slack, M. G. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical AI* 9(2–3):237–256.

Bouguet, J. 2000. Matlab camera calibration toolbox.

Brenner, M.; Hawes, N.; Kelleher, J.; and Wyatt, J. 2007. Mediating between qualitative and quantitative representations for task-orientated human-robot interaction. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*.

Brooks, R. A.; Breazeal, C.; Marjanovic, M.; Scassellati, B.; and Williamson, M. M. 1999. The Cog project: Building a humanoid robot. In *Computation for Metaphors, Analogy and Agents*, 52–87. Springer-Verlag.

Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S.; and Orebäck, A. 2005. Towards component-based robotics. In *Proceedings of International Conference on Intelligent Robots and Systems (IROS 2005)*.

Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S.; and Orebäck, A. 2007. Orca: A component model and repository. In Brugali, D., ed., *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics. Springer. 231–251.

Brooks, R. A. 1985. A robust layered control system for a mobile robot. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA.

Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* 47(1–3):139–159.

Bruyninckx, H. 2001. Open robot control software: The OROCOS project. In *Proceedings of International Conference on Robotics and Automation (ICRA)*, volume 3, 2523–2528.

Bruyninckx, H. 2008. The OROCOS project. http://www.orocos.org.

Buttazzo, G. C. 1997. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers.

Cabrera-Gámez, J.; Domínguez-Brito, A.; and Hernández-Sosa, D. 2001. CoolBOT: A component-oriented programming framework for robotics. In *Modelling of Sensor-Based Intelligent Robot Systems*. Springer. 282–304.

Carzaniga, A.; Rosenblum, D. R.; and Wolf, A. L. 1999. Challenges for distributed event services: Scalability vs. expressiveness. In *Engineering Distributed Objects '99*.

Carzaniga, A.; Rosenblum, D. S.; and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 19(3):332–383.

Chaimowicz, L.; Cowley, A.; Sabella, V.; and Taylor, C. 2003. ROCI: A distributed framework for multi-robot perception and control. In *Proceedings of IROS*, 266–271.

Chaudron, L.; Cossart, C.; Maille, N.; and Tessier, C. 1997. A purely symbolic model for dynamic scene interpretation. *International Journal on Artificial Intelligence Tools* 6(4):635–664.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, 300–307. AAAI Press.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 2000. *Model Checking*. The MIT Press.

Collett, T. H.; MacDonald, B. A.; and Gerkey, B. P. 2005. Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005)*.

Conte, G. 2007. *Navigation Functionalities for an Autonomous UAV helicopter*. Licentiate thesis, Linköpings universitet, Linköping. Linköping studies in science and technology. Thesis No. 1307.

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

Coste-Maniere, E., and Simmons, R. 2000. Architecture, the backbone of robotic systems. In *Proceedings of the IEEE Conference on Robotics and Automation*.

Cowley, A.; Chaimowicz, L.; and Taylor, C. 2006. Design minimalism in robotics programming. *International Journal of Advanced Robotic Systems* 3(1):31–36.

Cowley, A.; Hsu, H.; and Taylor, C. 2004. Distributed sensor databases for multi-robot teams. In *Proceeding of ICRA*.

C.R.S. website. Retrieved December 17, 2008, from http://crs.elibel.tm.fr.

Cugola, G.; Nitto, E. D.; and Fuggetta, A. 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.* 27(9):827–850.

Côté, C.; Letourneau, D.; Michaud, F.; Valin, J. M.; Brosseau, Y.; Raievsky, C.; Lemay, M.; and Tran, V. 2004. Code reusability tools for programming mobile robots. In *Proceedings of Intelligent Robots and Systems (IROS 2004)*, 1820–1825.

Côté, C.; Létourneau, D.; Michaud, F.; and Brosseau, Y. 2005. Software design patterns for robotics: Solving integration problems with marie. In *Proceedings of Workshop of Robotic Software Environment*.

Côté, C.; Brosseau, Y.; Létourneau, D.; Raïevsky, C.; and Michaud, F. 2006. Robotic software integration using MARIE. *International Journal of Advanced Robotic Systems* 3(1):55–60.

Côté, C.; Champagne, R.; and Michaud, F. 2007. Coping with architectural mismatch in autonomous mobile robotics. In *Proceedings Workshop on Software Development and Integration in Robotics: Understanding Robot Software Architectures, International Conference on Robotics and Automation*.

Côté, C.; Létourneau, D.; and Michaud, F. 2007. Robotics system integration frameworks: Marie's approach to software development and integration. In Brosseau, Y., ed., *Springer Tracts in Advanced Robotics: Software Engineering for Experimental Robotics*. Heidelberg: Springer Verlag.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In Cohn, A. G.; Schubert, L. K.; and Shapiro, S. C., eds., *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR-1998)*, 453–465. Morgan Kaufmann.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Doherty, P., and Kvarnström, J. 1999. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In Dixon, C., and Fisher, M., eds., *Proceedings of the 6th International Workshop on Temporal Representation and Reasoning (TIME-1999)*, 47–54. IEEE Computer Society Press.

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

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

Doherty, P., and Meyer, J.-J. C. 2007. Towards a delagation framework for aerial robotic mission scenarios. In *11th International Workshop on Cooperative Information Agents*, 5–26.

Doherty, P., and Rudol, P. 2007. A UAV search and rescue scenario with human body detection and geolocalization. In *20th Australian Joint Conference on Artificial Intelligence (AI-2007)*, LNCS, 1–13. Springer.

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

Doherty, P.; Gustafsson, J.; Karlsson, L.; and Kvarnström, J. 1998. TAL: Temporal Action Logics – language specification and tutorial. *Electronic Transactions on Artificial Intelligence* 2(3–4):273–306.

Doherty, P.; Granlund, G.; Kuchcinski, K.; Sandewall, E.; Nordberg, K.; Skarman, E.; and Wiklund, J. 2000. The WITAS unmanned aerial vehicle project. In Horn, W., ed., *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, 747–755. Amsterdam, The Netherlands: IOS Press.

Doherty, P.; Haslum, P.; Heintz, F.; Merz, T.; Nyblom, P.; Persson, T.; and Wingman, B. 2004. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 221–230.

Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Automated Agents and Multi-Agent Systems* Forthcoming.

Doherty, P.; Łukaszewicz, W.; and Szałas, A. 1995. Computing circumscription revisited: Preliminary report. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, volume 2, 1502–1508. Morgan Kaufmann.

Doherty, P.; Łukaszewicz, W.; and Szałas, A. 1997. Computing circumscription revisited: A reduction algorithm. *Journal of Automated Reasoning* 18:297–336.

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

Doherty, P. 1994. Reasoning about action and change using occlusion. In Cohn, A. G., ed., *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-1994)*, 401–405. Chichester, England: John Wiley and Sons.

Doherty, P. 2004. Advanced research with autonomous unmanned aerial vehicles. In Dubois, D.; Welty, C. A.; and Williams, M.-A., eds., *Proceedings on the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR-2004)*. AAAI Press. Extended abstract for plenary talk.

Doherty, P. 2005. Knowledge representation and unmanned aerial vehicles. In Skowron, A.; Barthès, J.-P. A.; Jain, L. C.; Sun, R.; Morizet-Mahoudeaux, P.; Liu, J.; and Zhong, N., eds., *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005)*, 9–16. IEEE Computer Society.

Domínguez-Brito, A. C.; Hernández-Sosa, D.; Isern-González, J.; and Cabrera-Gámez, J. 2007. CoolBOT: A component model and software infrastructure for robotics. In Brugali, D., ed., *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics. Springer. 143–168.

Dousson, C. 2002. Extending and unifying chronicle representation with event counters. In *Proceedings of ECAI'02*.

Drusinsky, D. 2003. Monitoring temporal rules combined with time series. In *Proceedings of the Computer Aided Verification Conference (CAV-2003)*, volume 2725 of *LNCS*, 114–118. Springer-Verlag.

Duranti, S.; Conte, G.; Lundström, D.; Rudol, P.; Wzorek, M.; and Doherty, P. 2007. LinkMAV, a prototype rotary wing micro aerial vehicle. In *Proceedings of the 17th IFAC Symposium on Automatic Control in Aerospace (ACA-2007)*.

Edsinger-Gonzales, A., and Weber, J. 2004. Domo: A force sensing humanoid robot for manipulation research. In *Proceedings of 4th IEEE/RAS International Conference on Humanoid Robots*, 273–291.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*. Elsevier and MIT Press. 997–1072.

Emmerich, W. 2000. Software engineering and middleware: A roadmap. In Finkelstein, A., ed., *The Future of Software Engineering*. ACM Press.

Estlin, T.; Rabideau, G.; Mutz, D.; and Chien, S. 2000. Using continuous planning techniques to coordinate multiple rovers. *Electronic Transactions on Artificial Intelligence* 5(16). http://www.ep.liu.se/ea/cis/2000/016/.

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

Farinelli, A.; Grisetti, G.; and Iocchi, L. 2005. SPQR-RDK: A modular framework for programming mobile robots. In *RoboCup 2004: Robot Soccer World Cup VIII*, 653–660.

Farinelli, A.; Grisetti, G.; and Iocchi, L. 2006. Design and implementation of modular software for programming mobile robots. *International Journal of Advanced Robotic Systems* 3(1):37–42.

Fernádez-Pérez, J. L.; Domínguez-Brito, A. C.; Hernández-Sosa, D.; and Cabrera-Gámez, J. 2004. Integrating systems in robotics. In *Proceedings of Conference on Robotics, Automation and Mechatronics*.

Fernández, J. L., and Simmons, R. G. 1998. Robust execution monitoring for navigation plans. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-1998)*, 551–557.

Fernyhough, J.; Cohn, A.; and Hogg, D. 1998. Building qualitative event models automatically from visual input. In *Proceedings of the International Conference on Computer Vision ICCV98*.

Fichtner, M.; Grossmann, A.; and Thielscher, M. 2003. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae* 57(2–4):371–392.

Fikes, R. 1971. Monitored execution of robot plans produced by STRIPS. In *Proceedings of the IFIP Congress (IFIP-1971)*, 189–194.

Finkbeiner, B., and Sipma, H. 2004. Checking finite traces using alternating automata. *Formal Methods in System Design* 24(2):101–127.

Finzi, A.; Ingrand, F.; and Muscettola, N. 2004. Robot action planning and execution control. In *Proceedings of the 4th International Workshop on Planning and Scheduling for Space (IWPSS-2004)*.

FIPA. 2002. Foundation for intelligent physical agents (FIPA) ACL message structure specification. http://www.fipa.org/.

Fitzpatrick, P.; Metta, G.; and Natale, L. 2008. Towards long-lived robot genes. *Robotics and Autonomous Systems* 56(1):29–45.

Fleury, S.; Herrb, M.; and Chatila, R. 1997. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *International Conference on Intelligent Robots and Systems*, 842–848.

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

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gat, E.; Slack, M. G.; Miller, D. P.; and Firby, R. J. 1990. Path planning and execution monitoring for a planetary rover. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, 20–25. IEEE Computer Society Press.

Gerkey, B.; Vaughan, R.; and Howard, A. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceeding of the International Conference on Advanced Robotics (ICAR)*, 317–323.

Gertler, J. 1998. *Fault Detection and Diagnosis in Engineering Systems*. CRC Press.

Ghallab, M. 1996. On chronicles: Representation, on-line recognition and learning. In Aiello, L. C.; Doyle, J.; and Shapiro, S., eds., *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-1996)*, 597–607. San Francisco: Morgan Kaufmann.

Ghanem, N.; DeMenthon, D.; Doermann, D.; and Davis, L. 2004. Representation and recognition of events in surveillance video using petri nets. In *Proceedings of Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'04)*.

Gore, P.; Schmidt, D. C.; Gill, C.; and Pyarali, I. 2004. The design and performance of a real-time notification service. In *Proc. of the 10th IEEE Real-time Technology and Application Symposium*.

Gruber, R.; Krishnamurthy, B.; and Panagos, E. 2001. CORBA notification service: Design challenges and scalable solutions. In *17th International Conference on Data Engineering*, 13–20.

Gustafsson, J., and Kvarnström, J. 2004. Elaboration tolerance through object-orientation. *Artificial Intelligence* 153:239–285.

Gustafsson, T. 2007. *Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems*. Ph.D. Dissertation, Linköpings universitet. Linköping Studies in Science and Technology, Dissertation No 1120.

Haigh, K. Z., and Veloso, M. M. 1998. Planning, execution and learning in a robotic agent. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems 1998 (AIPS-1998)*, 120–127. AAAI Press.

Hall, D. 1992. *Mathematical Techniques in Multisensor Data Fusion*. Artech House.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3):231–274.

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

Harrison, T.; Levine, D.; and Schmidt, D. C. 1997. The design and performance of a real-time CORBA event service. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, 184–200. New York: ACM Press.

Hawes, N.; Sloman, A.; Wyatt, J.; Zillich, M.; Jacobsson, H.; Kruijff, G.-J.; Brenner, M.; Berginc, G.; and Skočaj, D. 2007. Towards an integrated robot with multiple cognitive functions. In *AAAI*, 1548–1553. AAAI Press.

Hawes, N.; Wyatt, J.; and Sloman, A. 2006. An architecture schema for embodied cognitive systems. Technical Report CSR-06-12, University of Birmingham, School of Computer Science.

Hawes, N.; Zillich, M.; and Wyatt, J. 2007. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of IEEE RO-MAN 2007*, 998–1003.

Heimbigner, D., and Mcleod, D. 1985. A federated architecture for information management. *ACM Trans. Inf. Syst.* 3(3):253–278.

Heintz, F., and Doherty, P. 2004a. DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems* 15(1):3–13.

Heintz, F., and Doherty, P. 2004b. Managing dynamic object structures using hypothesis generation and validation. In *Proceedings of the AAAI Workshop on Anchoring Symbols to Sensor Data*.

Heintz, F., and Doherty, P. 2005a. DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In Dunin-Keplicz, B.; Jankowski, A.; Skowron, A.; and Szczuka, M., eds., *Monitoring, Security, and Rescue Techniques in Multiagent Systems*, Advances in Soft Computing, 479–492. Heidelberg: Springer Verlag.

Heintz, F., and Doherty, P. 2005b. A knowledge processing middleware framework and its relation to the JDL data fusion model. In Blasch, E., ed., *Proceedings of the Eighth International Conference on Information Fusion (Fusion'05)*.

Heintz, F., and Doherty, P. 2005c. A knowledge processing middleware framework and its relation to the JDL data fusion model. In Ögren, P., ed., *Proceedings of the Swedish Workshop on Autonomous Robotics (SWAR'05)*.

Heintz, F., and Doherty, P. 2006. DyKnow: A knowledge processing middleware framework and its relation to the JDL data fusion model. *Journal of Intelligent and Fuzzy Systems* 17(4):335–351.

Heintz, F., and Doherty, P. 2008. DyKnow federations: Distributing and merging information among UAVs. In *Proceedings of the 11th International Conference on Information Fusion (Fusion'08)*.

Heintz, F.; Kvarnström, J.; and Doherty, P. 2008a. Bridging the sense-reasoning gap: DyKnow – A middleware component for knowledge processing. In *Proceedings of the IROS workshop on Current software frameworks in cognitive robotics integrating different computational paradigms*.

Heintz, F.; Kvarnström, J.; and Doherty, P. 2008b. Knowledge processing middleware. In Carpin, S.; Noda, I.; Pagello, E.; Reggiani, M.; and von Stryk, O., eds., *Proceedings of the first international conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, volume 5325 of *LNAI*, 147–158. Springer Verlag.

Heintz, F.; Rudol, P.; and Doherty, P. 2007a. Bridging the sense-reasoning gap using DyKnow: A knowledge processing middleware framework. In Hertzberg, J.; Beetz, M.; and Englert, R., eds., *KI 2007: Advances in Artificial Intelligence*, volume 4667 of *LNAI*, 460–463. Springer Verlag.

Heintz, F.; Rudol, P.; and Doherty, P. 2007b. From images to traffic behavior – A UAV tracking and monitoring application. In *Proceedings of the 10th International Conference on Information Fusion (Fusion'07)*.

Heintz, F. 2001. Chronicle recognition in the WITAS UAV project – A preliminary report. In *SAIS 2001, Working notes*.

Henning, M. 2004. A new approach to object-oriented middleware. *IEEE Internet Computing* 66–75.

Huang, T.; Koller, D.; Malik, J.; Ogasawara, G.; Rao, B.; Russell, S.; and Weber, J. 1994. Automatic symbolic traffic scene analysis using belief networks. In *Proceedings of the 12th National Conference on Artificial intelligence*.

Huston, S. D.; Johnson, J. C. E.; and Syyid, U. 2003. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison-Wesley Professional.

Ingrand, F.; Lacroix, S.; Lemai-Chenevier, S.; and Py, F. 2007. Decisional autonomy of planetary rovers. *J. Field Robot.* 24(7):559–580.

JDL Data Fusion Subgroup. 1987. Data fusion lexicon.

Jensen, C. S.; Dyreson, C. E.; Böhlen, M.; Clifford, J.; Elmasri, R.; Gadia, S. K.; Grandi, F.; Hayes, P.; Jajodia, S.; Käfer, W.; Kline, N.; Lorentzos, N.; Mitsopoulos, Y.; Montanari, A.; Nonen, D.; Peressi, E.; Pernici, B.; Roddick, J. F.; Sarda, N. L.; Scalas, M. R.; Segev, A.; Snodgrass, R. T.; Soo, M. D.; Tansel, A.; Tiberio, P.; and Wiederhold, G. 1998. The consensus glossary of temporal database concepts — February 1998 version. In *Temporal Databases: Research and Practice*. Springer.

Kalman, R. E. 1960. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering* 82(Series D):35–45.

Kang, K. D.; Son, S.; and Stankovic, J. 2004. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering* 16(10):1200–1216.

Karlsson, L., and Gustafsson, J. 1999. Reasoning about concurrent interaction. *Journal of Logic and Computation* 9(5):623–650.

Kaupp, T.; Brooks, A.; Upcroft, B.; and Makarenko, A. 2007. Building a software architecture for a human-robot team using the orca framework. In *Proceeding of ICRA*, 3736–3741.

Kavraki, L. E.; Švestka, P.; Latombe, J.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Konolige, K.; Myers, K.; Ruspini, E.; and Saffiotti, A. 1997. The Saphira architecture: A design for autonomy. *J. Experimental and Theoretical AI* 9(2–3):215–235.

Koubarakis, M. 1994. Complexity results for first-order theories of temporal constraints. In Doyle, J.; Sandewall, E.; and Torasso, P., eds., *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR-1994)*, 379–390. Morgan Kaufmann Publishers, San Francisco, California, USA.

Kraetzschmar, G.; Utz, H.; Sablatnög, S.; Enderle, S.; and Palm, G. 2002. Miro – middleware for cooperative robotics. In Birk, A.; Coradeschi, S.; and Tadokoro, S., eds., *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *LNAI*. Berlin Heidelberg: Springer-Verlag.

Kramer, J., and Scheutz, M. 2007. Development environments for autonomous mobile robots: A survey. *Autonomous Robots* 22(2):101–132.

Krüger, D.; van Lil, I.; Sünderhauf, N.; R., B.; and Protzel, P. 2006. Using and extending the Miro middleware for autonomous mobile robots. In *Proceedings of Towards Autonomous Robotic Systems (TAROS)*, 90–95.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-2000)*, 995–1001.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

Kvarnström, J.; Heintz, F.; and Doherty, P. 2008. A temporal logic-based planning and execution monitoring system. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*.

Kvarnström, J. 2002. Applying domain analysis techniques for domain-dependent control in TALplanner. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, 101–110. AAAI Press, Menlo Park, California, USA.

Kvarnström, J. 2005. *TALplanner and Other Extensions to Temporal Action Logic*. Ph.D. Dissertation, Linköpings universitet. Linköping Studies in Science and Technology, Dissertation no. 937.

Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *Proceedings of the 19th National Conference of Artificial Intelligence (AAAI-2004)*, 617–622. AAAI Press.

Llinas, J.; Bowman, C.; Rogova, G.; Steinberg, A.; Waltz, E.; and White, F. 2004. Revisions and extensions to the JDL data fusion model II. In Svensson, P., and Schubert, J., eds., *Proc. of the 7th Int. Conf. on Information Fusion*.

Lu, C.; Stankovic, J. A.; Tao, G.; and Son, S. H. 2002. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-time Systems* 23(1–2):85–126.

Luckham, D. C. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Lyons, D., and Arbib, M. 1989. A formal model of computation for sensory-based robotics. *Robotics and Automation, IEEE Transactions on* 5(3):280–293.

Makarenko, A.; Brooks, A.; and Kaupp, T. 2006. Orca: Components for robotics. In *Proceedings of IROS Workshop on Robotic Standardization*.

Makarenko, A.; Brooks, A.; and Kaupp, T. 2007. On the benefits of making robotic software frameworks thin. In *Proceedings of IROS Workshop on Evaluation of Middleware and Architectures*.

Mallet, A.; Kanehiro, F.; Fleury, S.; and Herrb, M. 2007. Reusable robotics software collection. In *Proceedings of Workshop on Principles and Practice of Software Development in Robotics*.

Mallet, A.; Fleury, S.; and Bruyninckx, H. 2002. A specification of generic robotics software components: future evolutions of genom in the orocos context. In *Proceeding of International Conference on Intelligent Robots and Systems (IROS)*, 2292–2297.

Mantegazza *et. al.*, P. 2000. RTAI: Real time application interface. *Linux Journal* 72.

Markey, N., and Raskin, J.-F. 2006. Model checking restricted sets of timed paths. *Theoretical Computer Science* 358(2–3):273–292.

Markey, N., and Schnoebelen, P. 2003. Model checking a path. *CONCUR 2003 - Concurrency Theory* 251–265.

Medioni, G.; Cohen, I.; Bremond, F.; Hongeng, S.; and Nevatia, R. 2001. Event detection and analysis from video streams. *IEEE Trans. Pattern Anal. Mach. Intell.* 23(8):873–889.

Merz, T.; Duranti, S.; and Conte, G. 2004. Autonomous landing of an unmanned aerial helicopter based on vision and inertial sensing. In *Proceedings of the 9th International Symposium on Experimental Robotics (ISER-2004)*.

Merz, T.; Rudol, P.; and Wzorek, M. 2006. Control System Framework for Autonomous Robots Based on Extended State Machines. In *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS-2006)*.

Merz, T. 2004. Building a System for Autonomous Aerial Robotics Research. In *Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV-2004)*. Elsevier.

Metta, G.; Fitzpatrick, P.; and Natale, L. 2006. Yarp: Yet another robot platform. *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics* 3(1).

Michaud; F.; Cote; C.; Letourneau; D.; Brosseau; Y.; Valin; M; Beaudry; E.; Raievsky; C.; Ponchon; A.; Moisan; P.; Lepage; P.; Morin; Y.; Gagnon; F.; Giguere; P.; Roux; A; Caron; S.; Frenette; P.; Kabanza; and F. 2007. Spartacus attending the 2005 aaai conference. *Autonomous Robots* 22(4):369–383.

Mohamed, N.; Al-Jaroodi, J.; and Jawhar, I. 2008. Middleware for robotics: A survey. In *Proceedings of International Conference on Robotics, Automation, and Mechatronics*.

Montemerlo, M., and Thrun, S. 2007. *FastSLAM: A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics (Springer Tracts in Advanced Robotics)*. Berlin and Heidelberg: Springer-Verlag.

Motwani, R.; Widom, J.; Arasu, A.; Babcock, B.; Babu, S.; Datar, M.; Manku, G.; Olston, C.; Rosenstein, J.; and Varma, R. 2003. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*.

Myers, K. 1999. CPEF: Continuous planning and execution framework. *AI Magazine* 20(4):63–69.

Nagel, H.; Gerber, R.; and Schreiber, H. 2002. Deriving textual descriptions of road traffic queues from video sequences. In *Proc. 15th European Conference on Artificial Intelligence (ECAI-2002)*.

Nebel, B., and Burckert, H. J. 1995. Reasoning about temporal relations: A maximumal tractable subclass of allen's interval algebra. *Journal of ACM* 42(1):43–66.

Nesnas, I.; Simmons, R.; Gaines, D.; Kunz, C.; Diaz-Calderon, A.; Estlin, T.; Madison, R.; Guineau, J.; McHenry, M.; Shu, I.; and Apfelbaum, D. 2006. CLARAty: Challenges and steps toward reusable robotic software. *International Journal of Advanced Robotic Systems* 3(1):23–30.

Nesnas, I. 2007. The CLARAty project: Coping with hardware and software heterogeneity. In Brugali, D., ed., *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics. Springer.

Nii, H. P.; Feigenbaum, E. A.; Anton, J. J.; and Rockmore, A. J. 1988. Signal-to-symbol transformation: HASP/SIAP case study. In *Readings from the AI magazine*. Menlo Park, CA: AAAI.

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

Object Management Group. 2005. The real-time CORBA specification v 1.2.

Object Management Group. 2007. The data distribution service specification v 1.2.

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

Object Web. 2003. The object web website. Retrieved December, 2008, from http://middleware.objectweb.org/.

Orebäck, A., and Christensen, H. 2003. Evaluation of architectures for mobile robotics. *Autonomous Robots* 14(1):33–49.

Pell, B.; Gamble, E. B.; Gat, E.; Keesing, R.; Kurien, J.; Millar, W.; Nayak, P. P.; Plaunt, C.; and Williams, B. C. 1998. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proceeding of Agents 1998*, 369–376.

Pettersson, O. 2005. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems* 53(2):73–88.

Pettersson, P. O. 2006. *Sampling-based path planning for an autonomous helicopter*. Licentiate thesis, Linköpings universitet. Linköping studies in science and technology. Thesis No. 1229.

Pietzuch, P. R. 2004. *Hermes: A Scalable Event-Based Middleware*. Ph.D. Dissertation, University of Cambridge.

Ramamritham, K.; Son, S. H.; and Dipippo, L. C. 2004. Real-time databases and data services. *Real-Time Syst.* 28(2–3):179–215.

Rosu, G., and Havelund, K. 2005. Rewriting-based techniques for runtime verification. *Automated Software Engineering* 12(2):151–197.

Rotenstein, A. M.; Rothenstein, A.; Robinson, M.; and Tsotsos, J. K. 2007. Robot middleware should support task-directed perception. In *Proc. International Conference on Robotics and Automation: 2nd Workshop on Software Development and Integration in Robotics*.

Rudol, P., and Doherty, P. 2008. Human body detection and geolocalization for UAV human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. In *Proceedings of the IEEE Aerospace Conference*, 1–8.

Rudol, P.; Wzorek, M.; Conte, G.; and Doherty, P. 2008. Micro unmanned aerial vehicle visual servoing for cooperative indoor exploration. In *Proceedings of the IEEE Aerospace Conference*.

Schantz, R., and Schmidt, D. C. 2006. Middleware for distributed systems. In Wah, B., ed., *Encyclopedia of Computer Science and Engineering*. Wiley.

Scheutz, M., and Kramer, J. 2006. RADIC – a generic component for the integration of existing reactive and deliberative layers for autonomous robots. In *Proceedings of the fifth international joint conference on Autonomous Agents and Multiagent Systems*, 488–490. ACM.

Scheutz, M. 2006. ADE: Steps toward a distributed development and runtime environment for complex robotic agent architectures. *Applied Artificial Intelligence* 20(2–4):275–304.

Schlenoff, C.; Albus, J.; Messina, E.; Barbera, A. J.; Madhavan, R.; and Balakrisky, S. 2006. Using 4D/RCS to address AI knowledge integration. *AI Magazine* 27(2):71–82.

Schmidt, D. C. 2002a. Adaptive and reflective middleware for distributed real-time and embedded systems. *Lecture Notes in Computer Science* 2491:282–293.

Schmidt, D. C. 2002b. Middleware for real-time and embedded systems. *Communications of the ACM* 45(6):43–48.

Segall, B., and Arnold, D. 1997. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97, Brisbane Australia*.

Shanker, U.; Misra, M.; and Sarje, A. 2008. Distributed real time database systems: Background and literature review. *Distributed Parallel Databases* 23:127–149.

Shapiro, S. C., and Ismail, H. O. 1998. Embodied cassie. In *In Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium*, 136–143. AAAI Press.

Sheth, A. P., and Larson, J. A. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.* 22(3):183–236.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-1998)*, 1931–1937.

Soetens, P. 2007. The orocos component builder's manual v 1.4.1.

Steenstrup, M.; Arbib, M. A.; and Manes, E. G. 1983. Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences* 27:29–50.

Steinberg, A., and Bowman, C. 2001. Revisions to the JDL data fusion model. In Hall, D., and Llinas, J., eds., *Handbook of Multisensor Data Fusion*. CRC Press LLC.

Sun. 2000. Java remote method invocation specification. Technical report, Sun.

Thati, P., and Rosu, G. 2005. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science* 113:145–162.

The STREAM Group. 2003. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin, 26(1)*.

Thrun, S.; Montemerlo, M.; Dahlkamp, H.; Stavens, D.; Aron, A.; Diebel, J.; Fong, P.; Gale, J.; Halpenny, M.; Hoffmann, G.; Lau, K.; Oakley, C.; Palatucci, M.; Pratt, V.; Stang, P.; Strohband, S.; Dupont, C.; Jendrossek, L.-E.; Koelen, C.; Markey, C.; Rummel, C.; van Niekerk, J.; Jensen, E.; Alessandrini, P.; Bradski, G.; Davies, B.; Ettinger, S.; Kaehler, A.; Nefian, A.; and Mahoneyy, P. 2006. Stanley: The robot that won the darpa grand challenge: Research articles. *Journal of Field Robotics* 23(9):661–692.

Tsotsos, J. K. 1997. Intelligent control for perceptually attentive agents: The S* proposal. *Journal of Robotics and Autonomous Systems* 5–21.

Utz, H.; Sablatnög, S.; Enderle, S.; and Kraetzschmar, G. 2002. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation* 18(4):493–497.

Vaughan, R. T., and Gerkey, B. P. 2007. Reusable robot software and the player/stage project. In Brugali, D., ed., *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics. Springer. 267–289.

Vilain, M., and Kautz, H. 1986. Constraint propagation algorithms for temporal reasoning. In *Proc. AAAI'86*.

Volpe, P.; Nesnas, I.; Estlin, T.; Mutz, D.; Petras, R.; and Das, H. 2001. The CLARAty architecture for robotic autonomy. In *Proceedings of the 2001 IEEE aerospace conference*.

Washington, R.; Golden, K.; and Bresina, J. 2000. Plan execution, monitoring, and adaptation for planetary rovers. *Electronic Transactions on Artificial Intelligence* 5(17).

Wengert, C.; Reeff, M.; Cattin, P. C.; and Székely, G. 2006. Fully automatic endoscope calibration for intraoperative use. In *Bildverarbeitung für die Medizin*, 419–423. Springer-Verlag. http://www.vision.ee.ethz.ch/~cwengert/calibration_toolbox.php.

Weyhrauch, R. 1980. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence* 13(1–2):133–170.

White, F. 1988. A model for data fusion. In *Proc. of 1st National Symposium for Sensor Fusion*, volume 2.

Wilkins, D.; Lee, T.; and Berry, P. 2003. Interactive execution monitoring of agent teams. *Journal of Artificial Intelligence Research* 18:217–261.

Wzorek, M., and Doherty, P. 2006. Reconfigurable path planning for an autonomous unmanned aerial vehicle. In *Proceeding of ICAPS*.

Wzorek, M., and Doherty, P. 2009. A framework for reconfigurable path planning for autonomous unmanned aerial vehicles. *Journal of Applied Artificial Intelligence*. Forthcoming.

Wzorek, M.; Conte, G.; Rudol, P.; Merz, T.; Duranti, S.; and Doherty, P. 2006. From motion planning to control – a navigation framework for an autonomous unmanned aerial vehicle. In *Proceedings of the 21st Bristol International Unmanned Air Vehicle Systems (UAVS) Conference*. University of Bristol Department of Aerospace Engineering.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov***:* Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informa-

tionsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk**: Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN: 978-91-7393-792-4

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.