Retalis Language for Information Engineering in Autonomous Robot Software

Pouyan Ziafati[a,b*], Mehdi Dastani[b], John-Jules Meyer[b],
Leendert van der Torre[a,c] and Holger Voos[a]
[a] *Centre for Security, Reliability and Trust (SnT), University of Luxembourg*
[b] *Intelligent Systems Group, Utrecht University*
[c] *Computer Science and Communications Research Unit, University of Luxembourg*
{Pouyan.Ziafati, Leon.Vandertorre, Holger.Voos}@uni.lu
{M.M.Dastani, J.J.C.Meyer}@uu.nl

Abstract

Robotic information engineering is the processing and management of data to create knowledge of the robot's environment. It is an essential robotic technique to apply AI methods such as situation awareness, task-level planning and knowledge-intensive task execution. Consequently, information engineering has been identified as a major challenge to make robotic systems more responsive to real-world situations. The *Retalis* language integrates *ELE* and *SLR*, two logic-based languages. *Retalis* is used to develop information engineering components of autonomous robots. In such a component, *ELE* is used for temporal and logical reasoning, and data transformation in flows of data. *SLR* is used to implement a knowledge base maintaining a history of events. *SLR* supports state-based representation of knowledge built upon discrete sensory data, management of sensory data in active memories and synchronization of queries over asynchronous sensory data. In this paper, we introduce eight requirements for robotic information engineering, and we show how *Retalis* unifies and advances the state-of-the-art research on robotic information engineering. Moreover, we evaluate the efficiency of *Retalis* by implementing an application for a NAO robot. *Retalis* receives events about the positions of objects with respect to the top camera of NAO robot, the transformation among the coordinate frames of

We would like to thank three anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper. We would like to thank also Yehia El Rakaiby, Sergio Sousa, and Marc van Zee for their contributions in implementation or preparing the previous presentations of this work.

*Sponsored by *Fonds National de la Recherche Luxembourg (FNR).*

NAO robot, and the location of the NAO robot in the environment. About one thousand and nine hundreds events per second are processed in real-time to calculate the positions of objects in the environment.

1 Introduction

Robotic information engineering is the processing and management of data to create knowledge of the robot's environment. In artificial intelligence (AI), knowledge of the environment is typically represented in symbolic form. Symbolic representation of knowledge is essential for robots with AI capabilities such as situation awareness, task-level planning, knowledge-intensive task execution and human interaction [66, 14, 81, 72, 50]. Challenges of robotic information engineering include the processing and management of incremental, discrete and asynchronous sensory data such as recognized faces[1] [25], objects[2] [9], gestures [69] and behaviors [59]. Data processing includes applying logical, temporal, spatial and probabilistic reasoning techniques [71, 41, 16, 50, 46, 30, 65].

Both *on-demand* and *on-flow* processing of sensory data are necessary for a timely extraction and dissemination of information in robot software. On-demand processing is the modeling and management of data in different memory profiles such as short, episodic and semantic memories [79, 80, 65, 70]. Memory profiles are accessed and processed when required. For example, a plan execution component requests the location of a previously observed object in order to find it. On-flow processing is the processing of sensory data on the fly in order to extract information about the environment. An example is the monitoring of smoke and temperature sensor readings in order to detect fire. A fire alarm should be generated if there is smoke and the temperature is high, observed by sensors in close proximity within a given time interval. A notification about fire detection is sent, for instance, to a plan execution component to react on it. We refer to receivers of the notifications as consumers.

On-demand processing includes the following requirements.

1. Memorizing: data should be recorded selectively to avoid overloading memory.
2. Forgetting: outdated data should be pruned to bound the amount of recorded data in memory.
3. Active memory: memory should notify consumers when it is updated with relevant information. In this way, consumers can access the information at their time of convenience.

[1] http://wiki.ROS.org/face_recognition
[2] http://wiki.ROS.org/object_recognition

4. State-based representation: knowledge about the state of the robot's environment should be derived from discrete observations.

On-flow processing includes the following requirements.

1. Even-driven and incremental processing: on-flow processing requires a real-time event-driven processing model. Relevant information should be derived as soon as it can be inferred from the sensory data received so far. Therefore, sensory data should be processed and reasoned about as soon as they are received by the system. Moreover, real-time processing of sensory data requires incremental processing techniques.
2. Temporal pattern detection and transformation: on-flow processing requires detecting temporal patterns in flow of data and transforming data into suitable representations. The detection and transformation of data patterns are required to correlate and aggregate sensory data and detect high-level events occurring in the robot's environment.
3. Subscription: information derived from on-flow processing of data should be disseminated selectively. This is needed, for instance, not to overload a plan execution component with irrelevant events.
4. Garbage collection: records of data should be kept as far as they can contribute to derive relevant information and pruned afterwards. In the fire alarm example, a detection of smoke needs to be kept for a specified time period. If a relevant sensor detects a high temperature during this period, a fire alarm is generated. The record of the detected smoke is disregarded afterwards.

The aim of this paper is to support robotic information engineering. A key concern to develop affordable, maintainable and reliable robot software is the support of re-usability in development [19, 20, 35]. Support of information engineering includes identifying the requirements and providing re-usable solutions to requirements. Re-usability advances robot software by reducing development, maintenance and benchmarking costs [35, 54, 55, 42, 13, 37]. A robotic language should support information engineering as follows. First, it should support implementation at a suitable level of abstraction. This includes a qualitative representation of temporal relations among events as opposed to specifying such relations by occurrence times of events. Second, it should support efficient implementation, because an incremental processing and management of sensory data requires specialized algorithms and implementation care. Third, it should have clear semantics to support the correctness of implementation. In particular, the language semantics should take the asynchronicity of data into account. Fourth, it should support AI reasoning techniques as its built-in functionalities or by integration of relevant tools and libraries. For instance, logical and spatial reasoning capabilities are often necessary to rea-

son about the domain and common-sense knowledge, and spatial relations among objects.

Current systems do not support both on-flow and on-demand processing. The following examples illustrate the need to combine on-flow and on-demand processing. First, active memories generate events when the contents of their memories change. It is desirable that a consumer is able to subscribe for notification when a pattern of such changes occurs [80]. This requires an on-flow processing mechanism to process the memory events to detect relevant patterns of memory updates. Second, on-flow processing is needed for transforming data to a compact and suitable representation before recording it in memory. Third, simpler and more efficient implementation of some on-flow processing tasks can be achieved by mixing on-flow pattern recognition with on-demand querying of data in memory. In addition, on-flow and on-demand processing support of existing systems is limited. Open-source robotic software such as *ROS* [61] only facilitate flows of data among components. A state-of-the-art system is *DyKnow* [42, 38], which integrates multiple tools such as *C-SPARQL* [12] to support on-flow processing [27, 44, 39]. *C-SPARQL* does not support the expression of qualitative temporal relations among data or the filtering of data patterns based on their durations. Such capabilities are desirable, if not necessary, to capture complex data patterns [4]. In addition, on-flow processing in *DyKnow* requires semantic annotation of flows of data. Such semantic annotation is not provided in *ROS* software, widely used by the community, inducing programming overhead. Moreover, *DyKnow* is not available as open-source. Current on-demand processing systems often support either logical reasoning or active memory, but not both. An exception is the logic-based knowledge management system *ORO* [50, 49]. *ORO* supports active memory, but its support for the following on-demand requirements are limited. First, *ORO* does not support selectively memorizing data. All input data is recorded. Second, forgetting is limited to fixed memory profiles. It is not possible to specify forgetting policies based on types of data. Third, due to the open world assumption, representing and reasoning about dynamics of the robot's environment is difficult in *ORO*.

This paper introduces *Retalis* (*ETALIS*[3] [6, 5, 3] for Robotics) to supports on-flow and on-demand logical and temporal reasoning over sensory data and the state of robot's environment. *Retalis* is open source[4] and has been integrated in *ROS*. *Retalis* integrates the *Etalis* language for events (*ELE*)[5] for on-flow and develops the Synchronized Logical Reasoning language (*SLR*) [83] for on-demand processing.

[3] Event TrAnsaction Logic Inference System, http://code.google.com/p/etalis/

[4] https://github.com/procrob/Retalis

[5] *ETALIS* provides also the Event Processing SPARQL language (EP-SPARQL) [4] for event processing in Semantic Web applications.

By a seamless integration of these languages, *Retalis* supports the implementation of both on-flow and on-demand functionalities in one program.

The remainder of this paper is organized as follows. Section 2 presents a running example. Section 3 gives an overview of *Retalis*. Section 4 discusses on-flow processing requirements and describes *ELE*. Section 5 discusses on-demand processing requirements and presents *SLR*. Section 6 provides an evaluation of *Retalis*. Finally, Section 7 presents future work and concludes the paper.

2 Running Example

This section presents an example to illustrate the concepts and functionalities of *Retalis*. A robot is situated in a dynamic environment informing a person about the objects around it. The environment is described by a set of entities $e_1, e_2, ...$ These include the moving *base* of the robot, the pan-tilt 3D camera *cam* of the robot mounted on the *base*, a set of tables $table_1, table_2, ...$, a set of objects $o_1, o_2, ...$, a set of people $f_1, f_2, ...$, a set of attributes and a reference coordination frame *rcf*.

Figure 1 presents the robot's software components and their interactions. This figure should be read as follows. Directed arrows visualize asynchronous flows of data and two-way arrows represents request-response service calls. Asynchronous communications among the components are in the form of events. An event is a time-stamped piece of data formally defined in Section 4.1.

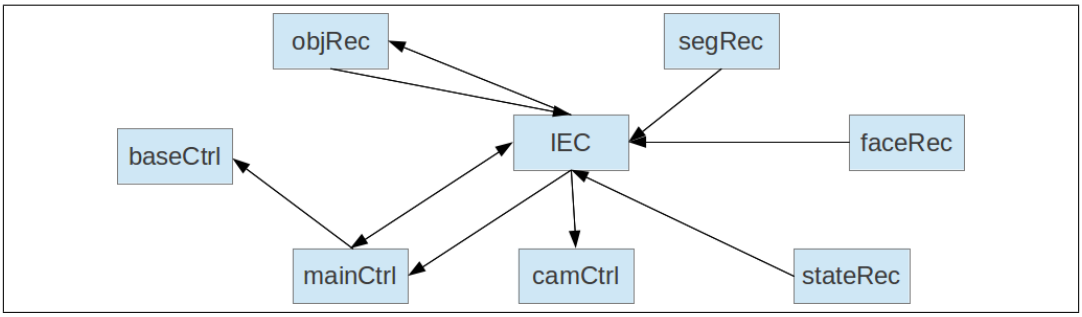

Figure 1: Robot's software components

The robot software includes the following components.

faceRec component: processes images from the camera, outputting $face(f_i, p_j)^t$ events. A $face(f_i, p_j)^t$ event represents the recognition of the face of f_i with confidence value p_j in a picture taken at time t.

segRec component: uses a real-time algorithm to process images from the camera into 3D point cloud data segments corresponding to individual objects. Such an algorithm is presented by Uckermann et al. [74]. The *segRec* component outputs $seg(o_i, c_j, p_k, l_g, pcl_h)^t$ events. Such an event represents the recognition of object o_i, with color c_j, with probability p_k, with relative position l_g to the *cam*, with the 3D point cloud data segment pcl_h recognized from a picture taken at time t. For events of the recognition of the same object segment over time, a unique identifier o_i is assigned using an anchoring and data association algorithm. Such an algorithm is presented by Elfring et al. [30].

objRec component: processes 3D point cloud data segments, outputting events of the form $obj(o_i, ot_j, p_k)^t$. Such an event represents the recognition of object type ot_j with probability p_k for object o_i recognized from a picture taken at time t.

stateRec component: localizes the robot. It outputs two types of events. A $tf(rcf, \text{base}, l_k)^t$ event represents the relative position between the reference coordination frame and the robot base at time t. A $tf(base, cam, l_k)^t$ event represents the relative position between the robot base and its camera at time t.

camCtrl and *baseCtrl* components: receive events of type $pos\_goal(l)$, each containing a position l to point the camera toward l or move the robot base to l, respectively.

IEC component: processes and manages events from *faceRec*, *segRec*, *stateRec* components. It detects reliable recognition of faces and objects and their movements to inform the *mainCtrl* component. Moreover, it positions objects in the reference coordination frame. In addition, it sends point cloud data of some objects to the *objRec* components to have their types recognized. The *IEC* component receives recognized types of objects from *objRec* as events and maintains the history of recognized faces and objects. It also controls the camera's position to follow a specific entity by sending perceived positions of the entity to *camCtrl*.

mainCtrl component: is responsible for interacting with the user. It moves the robot base by sending commands to the *baseCtrl* component. It receives events from *IEC* about the movements of objects to inform the user. The *mainCtrl* component queries *IEC* to answer the questions of the user.

3 Architectural Overview of *Retalis*

Retalis is a language for implementing Information Engineering Components (*IECs*) of autonomous robot systems. *IECs* are software components implementing a variety of information processing and management functionalities. *IEC*s are distributed independent components operating with other software components in parallel. *Retalis* does not impose any restriction on how components are structured in robot software.

Retalis represents and manipulates data as events. Events are time-stamped discrete pieces of data whose syntax is the same as *Prolog* ground terms [23, 53]. Events contain perceptual information such as a robot's position at a time or recognized objects in a picture. The meaning of events is domain-specific. The time-stamp of an event is a time point or a time interval referring to the occurrence time of the event. Events are time-stamped by the components generating them. [6] For example, the event $face(\text{'Neda'},70)^{28}$ could mean a recognition of Neda's face with 70% confidence in a picture taken at time 28 and the event $observed(\text{'Neda'})^{\langle 28,49 \rangle}$ could mean a frequent recognition of Neda's face in pictures taken during time interval [28,49]. An event containing information from processing of sensory data is usually time-stamped with the time at which the sensory data is acquired. This is usually different from the time point when the processing of the data is finished. A composite event generated from an occurrence of a pattern of other events is time-stamped based on the occurrence times of its composing events.

Retalis comprises two logic-based languages. The *ELE* language [6, 5, 3] supports on-flow processing and the *SLR* language [83] supports on-demand processing of data. In the *Retalis* program of an *IEC*, *ELE* generates composite events by detecting event patterns of interest in the input flow of events to the *IEC*. *SLR* is used to implement a knowledge base maintaining the history of some events. The knowledge base contains domain knowledge, including rules to reason about the recorded history. The flow of events processed by the *IEC* includes its input events and the composite events it generates. This means that composite events can in turn be used to detect other events. The robot software presented in Figure 1 includes one *IEC* component. Robot software can include a number of *IEC* components in order to modularize different information engineering tasks and to use distributed and parallel computing resources.

Figure 2 depicts the architecture of an *IEC*, including its logical components implemented in *Retalis*. This figure must be read as follows. Directed arrows visualize

[6] We assume all components share a central clock which is usually the clock of the computer running the components. If there is a network of computers running the components, time should be synchronized among them.

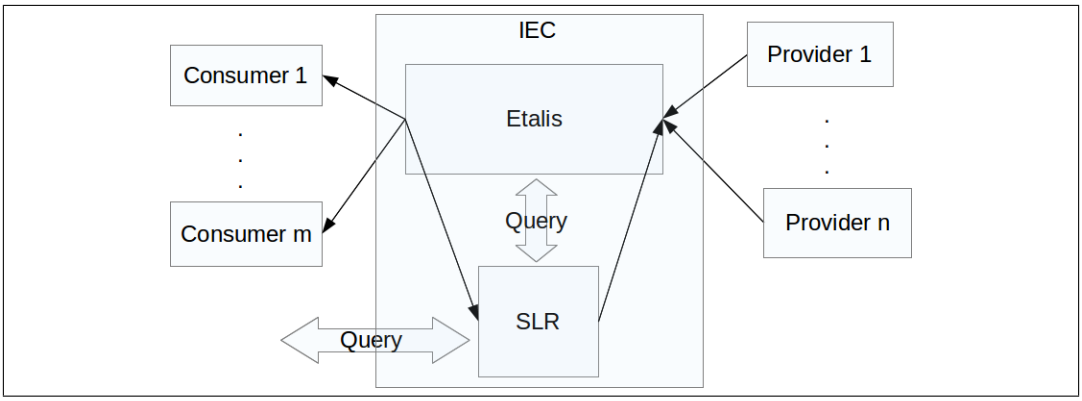

Figure 2: *IEC* architecture

asynchronous flows of events. Two-way arrows represent queries to *SLR* by *ELE* and external components.

Retalis supports the implementation of both synchronous and asynchronous interfaces among *IEC*s and other components. Asynchronous interaction is realized as follows. The *IEC* subscribes to events provided by *provider 1, .., provider n*. Moreover, *consumer 1, .., consumer n* subscribe to the *IEC* for types of events. The *Retalis* execution is event-driven. Input events are processed as they are received by the *IEC* to derive new events. When an event is processed, the event and resulting composite events are sent to interested consumers. The history of the input and derived events is also recorded in memory according to the *SLR* specification. *Retalis* specifications can be reconfigured at runtime. This includes the composite events to be detected, the producers the *IEC* is subscribed to, the subscriptions of consumers to the *IEC*, and the history of events maintained in memory.

Synchronous interactions between the *IEC* and other components are as follows. Components can query the domain knowledge and history of events in the *SLR* knowledge base. *Retalis* provides a request-response service to query *SLR*. *SLR* is a *Prolog*-based language, presented in Section 5.1 . The evaluation of a *SLR* query determines whether the query can be inferred from the knowledge base. The query evaluation may result in a variable substitution. The *IEC* can also access the functionalities of other software libraries or components. Function calls are supported both when answering queries and detecting composite events. To integrate external functionalities in *Retalis*, the corresponding software libraries should be interfaced with *Prolog*.

The interactions between *ELE* and *SLR* are as follows. On the one hand, *ELE* generates composite events. These events constitute the input flow of events to

SLR. *SLR* selectively records these events in its knowledge base. On the other hand, changes in the *SLR* knowledge base trigger corresponding input events for *ELE*. *ELE* can be used, for instance, to detect a pattern of such changes to inform the interested components. In addition, the specification of event patterns of interest in *ELE* can include queries to *SLR*. Queries are used to reason about the domain knowledge and history of events in *SLR*.

An *ELE* program, described in Section 4.1, contains two types of rules. The rules that include the ← symbol are event rules, specifying patterns of events to derive new events. The rules that include the :- symbol are static rules, constituting a *Prolog* program. The specification of the pattern of events in an event rule can include a query to the *Prolog* program defined by the static rules. *Retalis* programs are similar to *ELE* programs. The main difference is that the static rules in *Retalis* are *SLR* rules, constituting a *SLR* program which can be queries from the event rules.

Listing 1 presents an example of how *ELE* and *SLR* are used together in a *Retalis* program. This program records the position of the object segment o_1 whenever the position is changed by more than a meter. This program is read as follows. Capital letters represent variables. The body of the first and third rules are executed when the program is initialized. *c_mem(*m_1*,loc(*o_1*,L),*∞*,*∞*)* is a *SLR* clause creating memory m_1 recording the history of *loc(*o_1*,L)*^{T} events. The second rule is an *ELE* clause querying *SLR*, as written in its *WHERE* clause. For each *seg(*o_1*,C,P,L,PCL)*^{T} input event, the *prev* clause queries memory container m_1 for the last position of o_1 before time T. If the position has changed by more than a meter, the corresponding *loc(*o_1*,L)* event is generated and recorded in memory m_1. In addition, consumer *moving_objects* is notified by the corresponding event *obj(*o_1*)*^{T}. This is specified by the third rule, which is read as follows. The subscription s_1 subscribes consumer *moving_objects* to *loc(O,L)* events with the output template *obj(O)* from time 0. Details of the *ELE* and *SLR* languages are given in Sections 4.1 and 5.1.

```
1 onProgramStart :- c_mem(m1,loc(o1,L)^T,∞,∞).
2
3 loc(o1,L)^T <- seg(o1,C,P,L,PCL)^T
4                WHERE(
5                      prev(m1,loc(o1,L_prev)^T_prev,T)
6                      dist(L,L_prev,D),
7                      D > 1
8                     ).
9
```

```
10 onProgramStart :- sub(s1,moving_objects,loc(O,L),obj(O),0).
```

Listing 1: *Retalis* Program Example

A *Retalis* program is parsed and executed by a *Prolog* execution system and is provided a *C++* interface for communication with external components. This makes the *Retalis* language framework-independent, because its core depends only on a *Prolog* execution system. We use *SWI-Prolog*[7] [78] as the *Retalis* execution system and use the *SWI-Prolog C++ interface*[8] to interface the *SWI-Prolog* with *C++*. *Retalis* can be interfaced with existing robotic frameworks mapping its synchronous and asynchronous interfaces to their service-based and publish-subscribe communication mechanisms.

We have developed an interface to integrate *Retalis* with the *ROS* framework [61], the current de-facto standard in open-source robotics. In the *ROS* architecture, each *IEC* is a *ROS* component[9] [61]. Asynchronous and synchronous communications in *ROS* are realized using topics and services, respectively. By subscribing to a topic, a component receives the messages other components publish on that topic. A component invokes a service by sending a request message and receiving a response message.

Figure 3 presents an *IEC* in a *ROS* architecture. *IEC* is subscribed to Topics I_1 and I_2 receiving messages published by the components C_2 and C_3. *IEC* publishes events on topics O_1 and O_2 to which other components are subscribed.

To subscribe an *IEC* to a topic, the *Retalis-ROS* interface requires the name and message type of the topic. This is set in an *XML* configuration file, as in line 4-6 of Listing 2. The *Retalis-ROS* interface offers a number of services to reconfigure the *IEC* at runtime. These include services to subscribe the *IEC* to a topic, to unsubscribe from a topic and to subscribe a topic to events from the *IEC*. To publish an event on a *ROS* topic, the *Retalis-ROS* interface needs to know the message type of that topic. This can be set by the program, as in lines 7-9 of Listing 2, or at runtime.

[7] http://www.swi-prolog.org
[8] http://www.swi-prolog.org/pldoc/package/pl2cpp.html
[9] http://wiki.ROS.org/Nodes

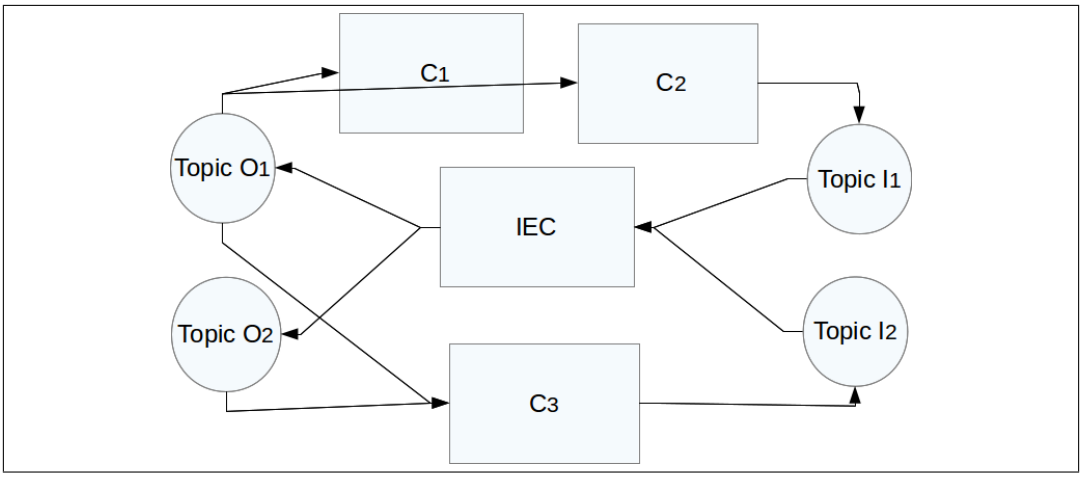

Figure 3: An *IEC* in *ROS* architecture

```
1  <?XML version="1.0"?>
2  <publish_subscribe>
3
4     <subscribe_to       name="/ar_pose_marker"
5                         msg_type="ar_pose/ARMarkers"
6     />
7     <publish_to         name="robot_marker_pos"
8                         msg_type="geometry_msgs/Transform"
9     />
10    <publish_to         name="gazeControl"
11                        msg_type="headTurn/GazeControl"
12    />
13
14 </publish_subscribe>
```

Listing 2: *Retalis-ROS XML* configuration file

The conversion among *ROS* messages and *Retalis* events is performed automatically by the *Retalis-ROS* interface. This may be described by an example. Table 1, consisting of five columns, depicts five standard *ROS* message types. The first row in each column is the name of a unique message type. The other rows presents the fields of data that the message type contains. Each field of a message contains a single datum or a list of data, whose type is a basic type such as Integer, Float, String, or it is a *ROS* message type. For example, a *geometry_msgs/Point* message contains three float values and a *geometry_msgs/Pose* message has a *geometry_msgs/Point* message as its first field of data.

geometry_msgs/PoseStamped	
std_msgs/Header	header
geometry_msgs/Pose	pose

std_msgs/Header	
uint32	seq
time	stamp
string	frame_id

geometry_msgs/Pose	
geometry_msgs/Point	p
geometry_msgs/Quaternion	o

geometry_msgs/Point	
float64	x
float64	y
float64	z

geometry_msgs/Quaternion	
float64	x
float64	y
float64	z
float64	w

Table 1: *ROS* message examples

Listing 3 presents the conversion of the *geometry_msgs/PoseStamped ROS* message type to its corresponding *Retalis* event. The conversion maps each *ROS* message to a *Prolog* compound term where the functor symbol of the term is the name of the message type and its arguments are the data fields of the message. Data of basic types such as Integer and Floats are represented by their values. Strings are wrapped by single quotes represented as *Prolog* Strings. Lists of data are represented as *Prolog* lists. Time in *ROS* is a basic data type expressed by two Integer values represented in a *Retalis* event as a list of two numbers.

When converting a *ROS* message to a *Retalis* event, the event is time-stamped with the time-stamp of the header of the message. If the message does not have a header, the event is time-stamped with the system current time. When converting a *Retalis* event to a *ROS* message, the time-stamp of the event is ignored. However, the *Retalis* language provides direct references to time-stamp of events. This can be used to set the *stamp* in the *std_msgs_____header(seq,stamp,frame_id)* argument of an event and hence in the header of its corresponding *ROS* message. *ROS* messages from different topics can be of the same type and need to be distinguished. Therefore, we encode topic names as main functor symbols of corresponding *Retalis* events. For example, if the event $p_n(t_1,..,t_n)^z$ is received from the topic x, the event is represented as $x(p_n(t_1,..,t_n))^z$.

```
1  geometry_msgs_____PoseStamped(
2              std_msgs_____Header(seq,stamp,frame_id),
3              geometry_msgs_____Pose(
4                                  geometry_msgs_____Point(x,y,z),
5                                     geometry_msgs_____Quaternion(x,y
                                          ,z,w)
6                                     )
7                                  )stamp
```

Listing 3: *Retalis* event format corresponding to geometry_msgs/PoseStamped *ROS* message type

4 On-Flow Information Processing

This section discusses on-flow processing requirements of robotic information engineering. It suggests the information flow processing systems [26], and in particular the *ELE* event-processing language [6, 5, 3], as suitable technologies to address the requirements. On-flow processing of data is widespread in large areas of robot software. As examples, we discuss in this section four robotic situations where on-flow processing of data is very useful.

The first situation is decoupling components interacting in robot software. This is usually supported by a publish-subscribe communication mechanism [31] based on an indirect addressing style [20, 80, 61, 42]. The publish-subscribe mechanism organizes robot software in a data-driven manner where components continuously process data generated by the other components. However, due to limited resources of a robot, sensory data needs to be processed selectively. This requires filtering of data passed among components. Data should be filtered based on the robot's operational context, such as its focus of attention. One way to support the filtering of data is to write complex software components whose processes can be reconfigured at runtime. However, such a reconfiguration might not be supported by the available components. The publish-subscribe support in most existing robotic frameworks is limited to topic-based interactions. Providers publish data items on topics, which are received by subscribers to those topics. In these frameworks, a component is usually subscribed to a fixed set of topics. More flexible and context-dependent interaction requires subscribers being able to specify their data of interest based on data patterns and policies [80, 42, 54]. Consider a robot looking for reliable recognition of yellow objects. The object segments sent to the object recognition component should be filtered to include only the yellow and reliably recognized object segments. Another example is the selective processing of new perceptions

of object segments by the object recognition component. A new perception of an object should be processed only when the object was perceived at a new location and this location did not change for a given time period.

The second situation is anchoring [24], creating symbolic representation of objects perceived from sensory data. The symbols and the data continuously sensed about the objects should be correlated. In an anchoring process, sensory data is interpreted into a set of hypotheses about recognized objects. For example, in a traffic monitoring scenario [42], images from color and thermal cameras are processed into a set of hypotheses about objects. The object hypotheses need to be correlated over time to deal with the data association problem [11]. There may be false positive and negative percepts, temporal occlusions of objects and visually similar objects in the environment. One can reason also about the hypotheses based on, for instance, the normative characteristics of the physical objects they represent [40, 30]. For example, in the traffic monitoring scenario, one can consider the positions and speeds of objects perceived over time and the layout of the road network. This can be used to reason about stationary and moving objects and their types. For instance, when a car is observed again after a temporary occlusion, it should be assigned the same symbol which was assigned to it previously.

The third situation pertains to flexible plan execution and monitoring in noisy and dynamic environments. The execution of actions/plans are to be driven, monitored and controlled by various conditions [76, 29, 81]. Conditions are monitored by low-level implementations of actions/behaviors to detect their success or failure. However, control and monitoring of plan execution via observation of various conditions at system-level is necessary. The advantages of system level plan execution control and monitoring are to use data provided by different perception components to achieve system's goals, to avoid complicating implementation of actions and to avoid duplicating monitoring functionalities. Depending on an application, conditions to be monitored can be as simple as monitoring an object for being attached to the manipulator. They can be also complex logical, temporal and numerical conditions.

The fourth situation is high-level event recognition to recognize and react in real-time to situations in the environment. One example is detecting traffic violations such as reckless driving by observing qualitative spatial relations among cars [43]. Another example is detecting situations and events such as "successful pass", "successful tackle" and "goal scoring" in football simulation or "washing hand before examination" and "basic clinical examinations carried out in time" in hospital simulations from lower level events [62]. The last example is recognizing human activities such as "cooking", "eating" and "watching TV" in smart homes [58, 66]. Detecting such situations of the environment requires correlating and aggregating sensory data

about changes of the environment based on their temporal and logical relations.

What all these situations have in common is a need for processing sensory data flow to extract new knowledge as soon as the relevant data becomes available without requiring persistent storage of data. Supporting on-flow processing requires an expressive and efficient language for real-time processing of data flows based on complex relations among the data items within the flows. On-flow processing is an important requirement in various application domains [26]. In environment monitoring, sensory data is processed to acquire information about the observed world, detect anomalies, or predict disasters. Financial applications analyze stock data to identify trends. Banking fraud detection and network intrusion detection require continuous processing of credit card transactions and network traffic, respectively. *RFID*-based inventory management requires continuous analysis of *RFID* readings. Manufacturing control systems often require observing system behavior to detect anomalies. As the result of many years of research from different research communities on such application domains, a large number of "information flow processing systems" have been developed to support on-flow processing of data [26].

An extensive survey of information flow processing systems [26] shows that the functionalities of these systems are converging to a set of operations and processing policies for on-flow filtering, combining and transformation of data, indicating universal usability of such functionalities for on-flow processing of data. This makes the existing information flow processing systems amenable to support on-flow information processing in robot software.

Current information flow processing research has led to two competing classes of systems [26], Data Stream Management Systems (*DSMS*s) and Complex Event-Processing Systems" (*CEPS*s). *DSMS*s functionalities resemble database management systems. They process generic flow of data through a sequence of transformations based on common *SQL* operators like selections, aggregates and joins. Being an extension of database systems, *DSMS*s focus on producing query answers, which are continuously updated to adapt to the constantly changing contents of their input data. In contrast, *CEPS*s see flowing data items as notification of events happening in the external world. These events should be filtered and combined to detect occurrences of particular patterns of events representing higher level events. *CEPS*s are rooted in publish-subscribe model. They increase the expressive power of subscribing language in traditional publish-subscribe systems with the ability to specify complex event patterns.

Both *DSMS*s and *CEPS*s have their own merits and the recent proposals attempt to combine the best of both classes of systems [26]. However, at this stage, the *CEPS*s are more suitable to support robotic on-flow processing due to the following reasons. First, the semantics given in *CEPS*s to data items as being event

notifications naturally corresponds to time-stamped sensory data being observations of the environment by the robot perception components. Second, *CEPS*s put great emphasis on detection and notification of complex patterns of events involving sequence and ordering relations which constitutes a large number of robotic on-flow information engineering problems which is usually out of the scope of *DSMS*s. The rest of this section introduces *ELE*, a state-of-the-art *CEPS*, and discusses its suitability for robotic on-flow information engineering through its comparison with related work.

4.1 *ETALIS* Language for Events (*ELE*)

ELE[10] [6, 5, 3] is an expressive and efficient language with formal declarative semantics for realizing complex event-processing functionalities. *ELE* advances the state-of-the-art *CEPSs* by allowing logical reasoning about domain knowledge in the specification of complex event patterns. Logical reasoning can be used to relate events, accomplish complex filtering and classification of events and enrich events on the fly with relevant background knowledge.

Event-processing functionalities in the *ELE* language are implemented by programming a set of static rules, encoding the domain knowledge and a set of event rules, specifying event patterns of interest to be detected in flow of data. The detected events can themselves match other event patterns, providing a flexible way of composing events in various steps of a hierarchy.

Definition 1 (*ELE* Signature [6]). A signature $\langle C, V, F_n, P^s_n, P^e_n \rangle$ for *ELE* language consists of:
- The set C of constant symbols.
- The set V of variables.
- For $n \in \mathbb{N}$ sets F_n of function symbols of arity n.
- For $n \in \mathbb{N}$ sets P^s_n of static predicate symbols of arity n.
- For $n \in \mathbb{N}$ sets P^e_n of event predicate symbols of arity n with typical elements p^e_n, disjoint from P^s_n.

Based on the *ELE* signature, the following notions are defined.

Definition 2 (Term [6]). A term $t ::= c \mid v \mid f_n(t_1, ..., t_n) \mid p^s_n(t_1, ..., t_n)$.

Definition 3 (Atom [6]). An static/event atom $a ::= p^{s/e}_n(t_1, ...t_n)$ where $p^{s/e}_n$ is a static/event predicate symbol and $t_1, ..., t_n$ are terms.

[10] http://code.google.com/p/etalis/

For example, the *face(F_i,P_j)* event atom is a template for observations of people's faces generated by the *faceRec* component.

Definition 4 (Event [6]). An event is a ground event atom time-stamped with an occurrence time.

- An atomic event refers to an instantaneous occurrence of interest.
- A complex event refers to an occurrence with duration.

For example, the occurrence time of the atomic event *face('Neda',70)*^{28} is time 28 and the occurrence time of the complex event *observed('Neda')*$^{\langle 28,49\rangle}$ is time interval $[28, 49]$.

Definition 5 (*ELE* Rule [6]). An *ELE* rule is a static rule r^s or an event rule r^e.

- A static rule is a Horn clause a :- $a_1, ..., a_n$ where $a, a_1, ..., a_n$ are static atoms. Static rules are used to encode the static knowledge of a domain.
- An event rule is a formula of the type $p^e(t_1, .., t_n) \leftarrow cp$ where cp is an event pattern containing all variables occurring in $p^e(t_1, .., t_n)$. An event rule specifies a complex event to be detected based on a temporal pattern of the occurrence of other events and the static knowledge.

Definition 6 (Event Pattern [6]). The language P of event patterns is

$$P ::= p^e(t_1, ..., t_n) \mid P \ \ WHERE \ \ t \mid q \mid (P).q \mid P \ \ BIN \ \ P \mid not(P).[P, P]$$

where p^e is an n-array event predicate, t_i denote terms, t is a term of type boolean, q is a non-negative rational number, and BIN is one of the binary operators *SEQ, AND, PAR, OR, EQUALS, MEETS, DURING, STARTS*, or *FINISHES*.

4.2 ELE Semantics

As opposed to most *CEPS*s, *ELE* has formal declarative semantics. The input to an *ELE* program is modeled as an event stream, a flow of events. The input event stream specifies that each atomic event occurs at a specific instance of time.

Definition 7 (Event Stream [6]). An event stream $\epsilon : Ground^e \rightarrow 2^{\mathbb{Q}^+}$ is a mapping from ground event atoms to sets of non-negative rational numbers.

For example, $\epsilon(obj(o, c, p)) = \{1, 3\}$ means among all events received by *ELE* as its input over its lifetime, the time points at which the event $objRec(o, c, p)$ occurs are 1 and 3.

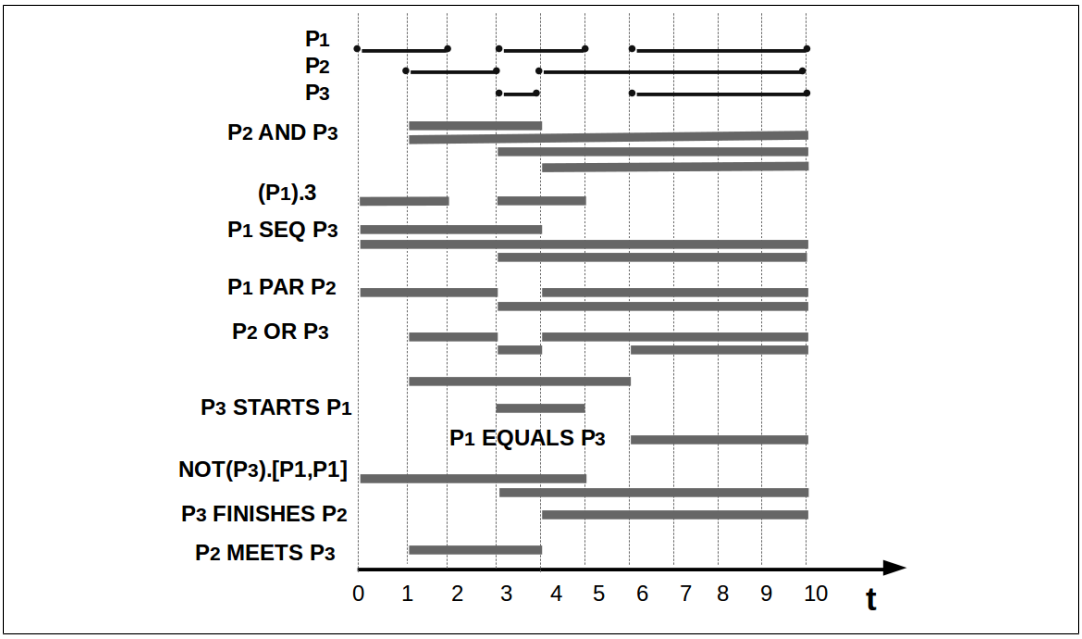

Figure 4: *ELE* event-processing operator examples, re-produced from [6]

Definition 8 (ELE semantics [6]). Given an ELE program with a set R of ELE rules, an event stream ϵ, an event atom a and two non-negative rational numbers q_1 and q_2, the ELE semantics determines whether an event $a^{\langle q_1,q_2\rangle}$, representing the occurrence of a with the duration $[q1, q2]$, can be inferred from R and ϵ (i.e. $\epsilon, R \models a^{\langle q_1,q_2\rangle}$).

Figure 4 informally introduces the *ELE* semantics. It provides examples of how *ELE* operators are used to specify complex events in terms of simpler ones. The first three lines show occurrences of the instances of events P_1, P_2 and P_3 during time interval [0,10]. The vertical dashed lines represent units of system time and horizontal bars represent detected complex events for the given patterns. The presented patterns are read as follows:

1. P_2 *AND* P_3: occurrence of both P_2 and P_3.
2. $(P_1).3$: occurrence of P_1 within an interval of length 3 time units.
3. P_1 *SEQ* P_3 : occurrence of P_3 after occurrence of P_1.
4. P_1 *PAR* P_2: occurrence of both P_1 and P_2 with non-zero overlap.

5. P_2 *OR* P_3: occurrence of P_2 or occurrence of P_3
6. P_1 *DURING* (1 *seq* 6): occurrence of P_1 during time interval [1,6]
7. P_3 *STARTS* P_1: occurrence of P_3 and P_1 both starting at the same time and P_3 ending earlier than P_1.
8. P_1 *EQUALS* P_3: occurrence of P_2 and P_3 both at the same time interval
9. $not(P_3).[P_1, P_1]$: occurrence of P_1 after occurrence of another P_1 where there is no occurrence of P_3 in between, during the end of the first P_1 and before the start of the second P_1.
10. P_3 *FINISHES* P_2: occurrence of P_3 and P_2 both ending at the same time and P_3 starting later than P_2.
11. P_2 *MEETS* P_3: occurrence of P_2 and P_3, P_3 starting at the exact time P_2 is ending.

For an example, consider the detection of fire from smoke and high temperature sensor readings. This task is implemented using the following *ELE* rule.

$$fireAlam \leftarrow$$
$$smoke(S1)\ AND\ high\_temperature(S2)$$
$$WHERE\ (\ nearby(S1, S2)\ ).$$

This rule is read as follows. *S1* and *S2* are variables. When smoke is detected by a sensor *S1* and high temprature is detected by a sensor *S2*, a fire alarm event is generated, if these sensors are located nearby. If *P2* and *P3* in figure 4 represent smoke and high-temperature events from sensors located nearby, then a fire alarm is generated four times during the time interval [0,10].

The static atom *nearby(S1,S2)* presents an example of logical reasoning in *ELE*. Given an ontology of sensors and their locations, this term specifies whether the sensors are located in the same area. Static atoms can be used to implement arbitrary functionalities in *Prolog*. In addition, they can be used as interface to foreign languages, for instance, to integrate libraries for spatial reasoning. In *Retalis*, *ELE* static terms are replaced by *SLR* queries to, in addition, reason about histories of events.

Complex events are time stamped based on the temporal patterns they represent. For example in Figure 4, the occurrence times of the first instances of *P2* and *P3* events are the intervals [1,3] and [3,4], respectively. According to *ELE* semantics, a

fire alarm detected from these events is time stamped with time interval [1,4]. The time stamp of detected patterns can be used to filter the patterns. For example, a fire alarm should be generated, only if both smoke and high temperature are detected within 300 seconds. This condition is added to the fire alarm pattern as follows.

$$
\begin{aligned}
fireAlam \leftarrow & \\
& (\\
& smoke(S1)\ AND\ high\_temperature(S2) \\
& WHERE\ (\ nearby(S1, S2)\ ) \\
&).300.
\end{aligned}
$$

Filters on time intervals of event patterns are important for garbage collection. If the fire alarm pattern does not contain the timing condition, a detection of smoke should be recorded forever in order to generate an alarm whenever a high-temperature is sensed. When the pattern includes the timing condition, the record is deleted after 300 seconds. After this time, the detection of smoke is no longer relevant, even if a high temperature is detected. Irrelevant records of events are automatically deleted by *ELE* garbage collection mechanisms.

ELE is free of operational side-effects, including the order among event rules and delayed or out of order arrival of input events. For example, the sequence pattern in Figure 4 detects three events during the time interval [0,10], no matter the order in which *ELE* receives *P1* and *P3* events.

Listing 4 presents an *ELE* program to illustrate the modeling capabilities of the *ELE* language. In this program, the robot detects an event whenever a person moves an object. Such an event is detected when a person's face is observed while the object is moved.

The program is read piece by piece. The first clause generates a $see(f)$ event for every two immediate consecutive recognitions of a face f, occurring with confidence values over fifty within half a second. The variable F is used to group the recognitions of faces in the event pattern and to pass information to generated events. The rule also explicitly encodes the start and end times of the sequence in content of the generated event by T_s and T_e variables.[11] The second clause detects reliable recognition of objects, when recognized three times within half a second with average confidence value over sixty. *pos_avg* is a static atom computing position of the object by averaging from its perceived positions. The third clause detects cases when an object is moved over five centimetres within a second. The fourth clause combines

[11] This is implemented by adding the $CHECK(t1(T_s), t2(T_e))$ clause which, for brevity, has been omitted.

each two overlapping movement events of an object into a new one with a longer occurrence time. The fifth clause combines two time periods of observing a person if they occur within three seconds after each other. Finally, the last clause detects when an object is moved during the time period a person is being observed.

```
1  see(F,Ts,Te) <-
2          (
3           NOT(face(F,P3)).[face(F,P1), face(F,P2)]
4           WHERE(P1 > 50, P2 > 50)
5          ).0.5s.
6
7  relSeg(O,L) <-
8          (
9           seg(O,C,P1,L1,X1)  SEQ  seg(O,C,P2,L2,X2)
10          SEQ  seg(O,C,P3,L3,X3)
11          WHERE(pos_avg([L1,L2,L3],L), avg([P1,P2,P3],P), P>60)
12         ).0.5s.
13
14 mov(O,L1,L2,Ts,Te) <-
15         (
16          relSeg(O,L1)  AND  relSeg(O,L2)
17          WHERE(dist([L_2,L_1],L), L>0.05)
18         ).1s.
19
20 mov(O,L1,L4,T1,T4) <-
21         mov(O,L1,L2,T1,T2)  PAR  mov(O,L3,L4,T3,T4)
22         WHERE(T3>T1).
23
24 see(F,T1,T4) <-
25          (see(F,T1,T2)  SEQ  see(F,T3,T4))
26          OR
27          (see(F,T1,T2) MEETS  see(F,T3,T4))
28          WHERE(T3-T2<3).
29
30 movBy(O,F,L2,T2) <-
31          mov(O,L1,L2,T1,T2)  DURING  see(F,T1,T2).
```

Listing 4: An *ELE* program for monitoring objects moved by humans

Assume an object has moved while the robot was seeing a face of a person. If the robot continues to see the face, the above rules generate more and more events indicating the person has moved the object, but one of such events might be sufficient for an application. Each time a new event occurs, the event along with the past events can match the pattern of a rule in several ways.

The *ELE* language offers various *consumption policies* to filter our repetitive rule firings. These includes policies to select a particular pattern among possible matches and to limit the use of an event to fire a rule more than once. While such policies are not aligned with declarative semantics of *ELE*, they are widely adopted in *CEPS*s for practical reasons. *ELE* also supports adding or deleting *ELE* rules at runtime allowing flexible reconfiguration of event-processing functionalities.

4.3 Runtime Subscription in *Retalis*

The *ELE* interface facilitates programming a fixed set of output channels to deliver certain types of events to consumers. *Retalis* extends this functionality enabling robot software components to subscribe to *Retalis* for their events of interest at run-time. The events are sent to subscribers asynchronously as soon as they are processed by *Retalis*.

A component subscribes to *Retalis* by sending a subscription request using a *ROS* service that the *Retalis* interface provides. A subscription is of the form $subscribe(Topic, Q, Tmpl, T_s, T_e)$. The process of the request by *Retalis* results in subscribing *Topic* to events matching the query pattern Q that have occurred during time interval $[T_s, T_e]$. A query pattern Q is a tuple $\langle e, Cond \rangle$, where e is an event atom and *Cond* is a set of conditions on variables which are arguments of e. An event P matches a query pattern Q when there is a substitution which can unify p and e and makes the conditions in *Cond* true (i.e. $\exists\theta(p = q_\theta)$).

When a subscription is registered, every event matching the subscription is asynchronously sent to the corresponding topic as the event is read from the *Retalis* input or generated by *ELE* rules. Events are first converted to the template form *Tmpl* before being sent to the topic. If a component does not know in advance the end time of its subscription, it can subscribe to its events of interest using $sub(Id, C, Q, Tmpl, T_s)$ and unsubscribe from them at any time using $unsub(Id, T_e)$. Id is a unique identifier of such a subscription.

Example 1. When the robot is asked to follow the object segment $seg11$, the control component sets the target location for the Gaze component to the location of $seg11$ by sending the following subscription command to the Information-

Engineering Component.

$$sub(100, \text{`}camCtrl\text{'}, \langle relObj(\text{`}seg11\text{'}, L), \langle\rangle\rangle, pos\_goal(L), \text{`}now\text{'})$$

Consequently, every time *IEC* processes an event $relObj(\text{`}seg11\text{'}, L)$, it sends the location $L$ of $seg11$ to the Gaze in the $pos\_goal(L)$ format. To unsubscribe, the control component sends the $unsub(100, \text{`}now\text{'})$ command to *IEC*.

4.4 Discussion

Previous robotic research is concerned with on-flow processing for specific research tasks such as component interaction, anchoring, monitoring and event-recognition. The consequence is the narrow scope of related robotic research reducing the community collaboration in supporting on-flow processing in robotic software. For instance, on-flow processing support of open-source robotic software such as *ROS* is limited to fixed publish-subscribe flow of data among components.

In parallel to this research, the *DyKnow* [42, 38] framework has been extended with a number of tools that are relevant to on-flow processing [27, 44, 39]. The main feature of the work is the annotation of data streams and transformation processes with semantic descriptions. The semantic descriptions are used for automatic construction of streams of data. The *C-SPARQL* [12] language has been integrated to support the querying of flows of data. *C-SPARQL* belongs to the *DSMS*s category of on-flow processing systems. The advantages of *ELE* over *C-SPARQL* is its support for capturing complex data patterns. In contrast, the *Retalis* does not support an automatic discovery of flows of data, for instance, required to detect a complex event. The input and output subscriptions of Information-Engineering Components and the event patterns they process are reconfigurable at runtime. However, such reconfigurations are not made automatic.

The literature does not contain a comparison between the expressive power of information flow processing systems. *ELE* is one of the most expressive systems as it supports most of the existing information flow processing operations listed in [26]. In particular, *ELE* supports the representation of all possible thirteen temporal relations between time interval occurrence times of two events as defined in Allen's interval algebra [2], non-occurrence of an event between the occurrence of two other events, and iterative and aggregating patterns. Furthermore, arbitrary processes can be applied on events through the use of static atoms in *ELE* syntax, provided that such processes are interfaced with the *Prolog* language. An example is interfacing spatial reasoning functionalities with *Prolog* presented in [72].

Logic-based approaches such as *Chronicle Recognition* [33] and *Event Calculus* [47, 67] have received considerable attention for event representation and recognition due to their merits, including expressiveness, formal and declarative semantics and being supported by machine learning tools to automate the construction and refinement of event recognition rules [8, 6]. However, the query-response execution mode and scalability of classic logic-based systems limits their usability for on-flow information processing. The query-response execution means detecting an event at runtime requires frequently querying the system for that event. Moreover, the event is detected only when the next time the system is queried for that event. In addition, efficient evaluation of such queries requires caching mechanisms not to re-evaluate queries over all historic data [22]. *ELE* bridges the gap between *CEPS*s and logic-based event-recognition systems by offering a logic-based *CEPS* with an event-driven, incremental and efficient execution model.

The *IDA* [80, 55] and *CAST* [37, 36] are robotic frameworks supporting the subscription of components to their events of interest based on the type and content of events. Using *XML* data format in *IDA*, a subscriber can register for information items containing specific field of data. *IDA* also provides few types of event filters such as the *Frequency filter*, which outputs only every n-th received notification. *Retalis* provides a general framework to address a much wider variety of event processing requirements, including temporal and spatial reasoning over events to detect complex event patterns. Moreover, the subscription mechanisms of *IDA* and *CAST* are tightly built over their underlying middleware. In contrast, *Retalis* is framework-independent and has been interfaced with *ROS* which is widely used by robotic community.

The use of *CEPS*s for detecting high-level events in agent research has been proposed before. Buford et al. [22] extend the *BDI* architecture with situation management components for event correlation in distributed large-scale systems. Ranathunga et al. [23] utilize the *ESPER*[12] event-processing language to detect high-level events in second life virtual environments.[13] However this work is not concerned with the robotic on-flow information-processing problem, it does not provide a formal account of event processing and does not support run-time subscription. Other related work includes various approaches for high-level event recognition, anchoring and monitoring, for instance, using Chronicle recognition, constraint satisfaction or variants of temporal logic [42, 58, 43, 28]. Such approaches do not satisfy all on-flow information processing requirements. For instance, the Chronicle recognition or constrained satisfaction approaches based on simple temporal networks cannot

[12] Esper Reference, Esper Team and EsperTech Inc, accessible at http://esper.codehaus.org/esper-4.9.0/doc/reference/en-US/html_single/

[13] http://secondlife.com

express atemporal constraints, and temporal logic based approaches do not support transformation of information.

5 On-Demand Information Processing

On-demand information processing corresponds to managing data in memory or knowledge base to be queried and reasoned upon on request. This section presents the *SLR* language[14] to address on-demand processing requirements related to discreteness, asynchronicity and continuity of robotic sensory data that are not satisfactorily supported by existing systems. After a short introduction of these requirements, the *SLR* syntax and semantics are presented and the usability of the language and its relation with existing works is discussed.

Building robot knowledge based on discrete observations is not always a straightforward task, since events contain various information types that should be represented and treated differently. For example, to accurately calculate the robot position at a time point, one needs to interpolate its value based on the discrete observations of its value in time. One also needs to deal with the persistence of knowledge and its temporal validity. For example, it might be reasonable to assume that the color of an object remains the same until a new observation is made indicating the change of color. In some other cases, it may not be safe to infer an information, such as the location of an object, based on an observation that is made in distant past. Building robot knowledge of its environment upon sensory events requires language support to simplify reasoning about the state of the environment at a time based on discrete observations of the environment.

A network of distributed and parallel components process robot sensory data and send the resulting events to the knowledge base. Due to processing times of the perception components and possible network delay, the knowledge base may receive the events with some delays and not necessarily in the order of their occurrence. For example, the event indicating the recognition of an object in a *3D* image is generated by the object recognition component sometime after the actual time at which the object is observed, because of object recognition processing time. Another example is when data is generated or needs to be verified by an external source with arbitrary operating time. Therefore, when the knowledge base is queried, correct evaluation of the query may require waiting for the perception components to finish processing of sensory data to ensure that all data necessary to evaluate the query is present in the knowledge base. For example, the query, "how many cups are on the table at time t?" should not be answered immediately at time t, but answering the query

[14] An earlier version of *SLR* is appeared in a technical report before [83].

should be delayed until after completing the processing of pictures of the table by the object recognition component and the reception of the results by the knowledge base. Dealing with asynchronicity of sensory data requires supporting the implementation of synchronization mechanisms to assure evaluating queries when relevant data to queries are available in the knowledge base.

Robot perception components continuously send their observations to the knowledge base, leading to a growth of memory required to store and maintain the robot knowledge. The unlimited growth of the event history leads to a degradation of the efficiency of query evaluation and may even lead to memory exhaustion. Bounding the growth of memory requires supporting the implementation of mechanisms to prune outdated data.

5.1 *SLR* Language for Event Management and Querying

Synchronized Logical Reasoning language (*SLR*) is a knowledge management and querying language for robotic software enabling the high-level representation, querying and maintenance of robot knowledge. In particular, *SLR* aims at simplifying the representation of robot knowledge based on its discrete and asynchronous observations and improving efficiency and accuracy of query evaluation by providing synchronization and event-history management mechanisms. These mechanisms facilitate ensuring that all data necessary to answer a query is gathered before the query is answered and that outdated and unnecessary data is removed from memory.

In an Information-Engineering Component programmed in *Retalis*, the input to *SLR* is the stream of events processed by *ELE*. This consists of the input stream of events to the *IEC*, time-stamped by the perception components and the events generated and time-stamped by *ELE*. The *SLR* language bears close resemblance to logic programming and is both in syntax and semantics very similar to *Prolog*. Therefore, we first review the main elements of *Prolog* upon which we define the *SLR* language.

In *Prolog* syntax, a *term* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a functor symbol and $t_1, \ldots, t_n$ are constants, variables or terms. A term is *ground* if it contains no variables. A *Horn clause* is of the form $a_1 \wedge \ \ldots \ \wedge a_n \rightarrow a$, where a is a term called the *Head* of the clause, and $a_1, \ldots, a_n$ is called the *Body* where a_i are terms or negation of terms. $a \leftarrow true$ is called a *fact* and usually written as a. A *Prolog program* P is a finite set of *Horn clauses*.

One executes a logic program by asking it a query. *Prolog* employs the *SLDNF* resolution method [7] to determine whether or not a query follows from the program. Given a goal, *SLDNF* tries to prove the goal using the rules and facts of the program.

A goal is proved if there is a variable substitution by applying which the goal matches a fact, or matches the head of a rule and the goals in body of the rule can be proved from left to right. Goals are resolved by trying the facts and rules in the order they appear in the program. A query may result in a substitution of free variables. We use $P \vdash_{SLDNF} Q\theta$ to denote a query Q on a program P, resulting in a substitution θ.

5.1.1 *SLR* Syntax

An *SLR* signature includes constant symbols, *Floating-point* numbers, variables, time points, and two types of functor symbols. Some functor symbols are ordinary *Prolog* functor symbols called *static functor symbols*, while the others are called *event functor symbols*.

Definition 9 (*SLR* Signature). A signature $S = \langle C, R, V, Z, P^s, P^e \rangle$ for *SLR* language consists of:
- A set C of *constant symbols*.
- A set $R \subseteq \mathbb{R}$ of *real numbers*.
- A set V of *variables*.
- A set $Z \subseteq R_{r \geq 0} \cup V$ of *time points*
- P^s, a set of P^s_n of *static functor symbols* of arity n for $n \in \mathbb{N}$.
- P^e, a set of P^e_n of *event functor symbols* of arity n for $n \in \mathbb{N}_{n \geq 2}$, disjoint with P^s_n.

Definition 10 (Term). A *static/event term* is of the form
$t ::= p^s_n(t_1, ..., t_n)/p^e_n(t_1, ..., t_{n-2}, z_1, z_2)$ where $p^s_n \in P^s_n$ and $p^e_n \in P^e_n$ are *static/event functor symbols*, t_i are *constant symbols*, *real numbers*, *variables* or *terms* themselves and z_1, z_2 are *time points* such that $z_1 \leq z_2$.

For the sake of readability, an event term is denoted as $p_n(t_1, \ldots, t_{n-2})^{[z_1, z_2]}$. Moreover, an event term whose z_1 and z_2 are identical is denoted as $p_n(t_1, \ldots, t_{n-2})^{z}$.

Definition 11 (Event). An *event* is a ground event term $p_n(t_1, ...t_n)^{[z_1, z_2]}$, where z_1 is called the *start time* of the event and z_2 is called its *end time*. The functor symbol p_n of an event is called its *event type*.[15]

We introduce two types of static terms, *next* and *prev* which respectively refer to occurrence of an event of a certain type observed right after and right before a time

[15] The representation of events in *SLR* and *ELE* is similar, but the *SLR* signature is defined in a way to be close to *Prolog*.

point, if such an event exists. In the next section we provide the semantics. In this section, we restrict ourselves to the syntax of *SLR*.

Definition 12 (Next Term). Given a signature S, a next term of the form $next(p_n(t_1, ...t_n)^{[z_1,z_2]}, z_s, z_e)$ has an $p_n(t_1, ...t_n)^{[z_1,z_2]}$ event term and two time points z_s, z_e representing a time interval $[z_s, z_e]$ as its arguments.

Definition 13 (Previous Term). Given a signature S, a previous term of the form $prev(p_n(t_1, ...t_n)^{[z_1,z_2]}, z_s)$ has an event term $p_n(t_1, ...t_n)^{[z_1,z_2]}$ and a time point z_s as its arguments.

Definition 14 (*SLR* Program). Given a signature S, an *SLR* program D consists of a finite set of Horn clauses of the form $a_1 \wedge \ldots \wedge a_n \rightarrow a$ built from the signature S, where *next* and *prev* terms can only appear in the body of rules and the program excludes event facts (i.e. events).

5.1.2 *SLR* Operational Semantics

An *SLR* knowledge base is modeled as an *SLR* program and an input stream of events. In order to limit the scope of queries on a *SLR* knowledge base, we introduce a notion of an event stream view, which contains all events occurring up to a certain time point.

Definition 15 (Event Stream). An *event stream* ϵ is a (possibly infinite) set of events.

Definition 16 (Event Stream View). An *event stream view* $\epsilon(z)$ is the maximum subset of event stream ϵ such that events in $\epsilon(z)$ have their end time before or at time point z, i.e. $\epsilon(z) = \{p_n(t_1, \ldots, t_{n-2})^{[z_1,z_2]} \in \epsilon \mid z_2 \leq z\}$.

Definition 17 (Knowledge Base). Given a signature S, a knowledge base k is a tuple $\langle D, \epsilon \rangle$ where D is an *SLR* program and ϵ is an event stream defined upon S.

Definition 18 (*SLR* Query). Given a signature S, an *SLR* query $\langle Q, z \rangle$ on an *SLR* knowledge base k consists of a regular *Prolog* query Q built from the signature S and a time point z. We write $k \vdash_{SLR} \langle Q, z \rangle \theta$ to denote an *SLR* query $\langle Q, z \rangle$ on a knowledge base k, resulting in a substitution θ.

The operational semantics of *SLR* for query evaluation follows the standard *Prolog* operational semantics (i.e. unification, resolution and backtracking) [7] as follows: The evaluation of a query $\langle Q, z \rangle$ given an *SLR* knowledge base $k = \langle D, \epsilon \rangle$

consists in performing a depth-first search to find a variable binding that enables derivation of Q from the rules and static facts in D, and events in ϵ. The result is a set of substitutions (i.e. variable bindings) θ such that $D \cup \epsilon \vdash_{SLDNF} Q\theta$ under the condition that event terms which are not arguments of *next* and *prev* terms can be unified with events that belonging to $\epsilon(z)$.

The event stream models observations made by robot perception components. Events are added to the *SLR* knowledge base in the form of facts when new observations are made. The z parameter of a query sets the scope of the query to set of observations made up until time z. This means that the query $\langle Q, z\rangle$ cannot be evaluated before time z, since *SLR* would not have received the robot's observations necessary to evaluate Q and the query can be evaluated as soon as all observations up to time z is in place. The only exceptions are the *prev* and *next* clauses whose evaluation might need observations made after time z. A query $\langle Q, z\rangle$ can be posted to *SLR* long after time z, in which case the *SLR* knowledge base contains observations made after time z. In order to have a clear semantics of queries, *SLR* evaluates a query $\langle Q, z\rangle$ by only taking into account the event facts in $\epsilon(z)$. Regardless of the z parameters of queries, the *next* or *prev* clauses are evaluated based on their declarative definitions as follows.

Definition 19 (Previous Term Semantics). A $prev(p_n(t_1, ...t_n)^{[z_1,z_2]}, z_s)$ term unifies $p_n(t_1, ...t_n)^{[z_1,z_2]}$ with an event $p_n(t'_1, ...t'_n)^{[z'_1,z'_2]}$ in $\epsilon(z_s)$ such that there is no other such event in $\epsilon(z_s)$ that has its end time later than z'_2. If such a unification is found, the *prev* clause succeeds and fails otherwise.

$$prev(p_n(t_1, ...t_n)^{[z_1,z_2]}, z_s) : \begin{cases} \theta & \exists p_n(t'_1, ...t'_n)^{[z'_1,z'_2]} \in \epsilon(z_s) | \\ & \quad \exists \theta((p_n(t_1, ...t_n)^{[z_1,z_2]})_\theta = (p_n(t'_1, ...t'_n)^{[z'_1,z'_2]})_\theta) \\ & \quad \wedge \nexists p_n(t"_1, ...t"_n)^{[z"_1,z"_2]} \in \epsilon(z_s) | \\ & \qquad z"_2 > z'_2 \ \wedge \\ & \qquad \exists \gamma((p_n(t_1, ...t_n)^{[z_1,z_2]}) \stackrel{\gamma}{=} \\ & \qquad\qquad (p_n(t"_1, ...t"_n)^{[z"_1,z"_2]})), \\ \text{fails} & \text{otherwise} \end{cases}$$

By definition, the variable z_s should be already instantiated when a *prev* clause is evaluated and an error is generated otherwise. It is also worth noting that a *prev* clause can be evaluated only after time z_s when all relevant events with end time earlier or equal to z_s have been received by and stored in the *SLR* knowledge base.

Definition 20 (Next Term Semantics). A $next(p_n(t_1, ...t_n)^{[z_1,z_2]}, z_s, z_e)$ term unifies $p_n(t_1, ...t_n)^{[z_1,z_2]}$ with an event $p_n(t'_1, ...t'_n)^{[z'_1,z'_2]}$ in $\epsilon(z_e)$ such that $z_s \leq z'_2 \leq z_e$ and there is no other such event in ϵ that has its end time earlier than z'_2. If such a

unification is found, the *next* clause succeeds and fails otherwise.

$$next(p_n(t_1,...t_n)^{[z_1,z_2]}, z_s, z_e) : \begin{cases} \theta & \exists p_n(t'_1,...t'_n)^{[z'_1,z'_2]} \in \epsilon(z_e)| \\ & \quad z'_2 \geq z_s \wedge \\ & \quad \exists\theta((p_n(t_1,...t_n)^{[z_1,z_2]})_\theta = (p_n(t'_1,...t'_n)^{[z'_1,z'_2]})_\theta) \\ & \quad \wedge\ \nexists p_n(t"_1,...t"_n)^{[z"_1,z"_2]} \in \epsilon(z_e)| \\ & \qquad z_s \leq z"_2 < z'_2 \wedge \\ & \qquad \exists\gamma((p_n(t_1,...t_n)^{[z_1,z_2]}) \stackrel{\gamma}{=} \\ & \qquad\qquad (p_n(t"_1,...t"_n)^{[z"_1,z"_2]})), \\ \text{fails} & \text{otherwise} \end{cases}$$

By definition, the variables z_s and z_e should be instantiated when a *next* clause is evaluated and an error is generated otherwise. A *next* clause can only be evaluated after time z_e when all relevant events with end time earlier or equal to z_e have been received and stored in the *SLR* knowledge base. However, if we assume that events of the same type (i.e. with same functor symbol and arity) are received by *SLR* in the order of their end times, the next clause can be evaluated as soon as *SLR* receives the first event with the end time equal or later than z_s which is unifiable with $p_n(t_1,...t_n)^{[z_1,z_2]}$, not to unnecessarily postpone queries.

The *next* and *prev* clauses can be implemented by the following two *Prolog* rules in which the $\neg$ symbol represents *Negation as failure*. However, we take advantage of the fact that *SLR* usually receives events of the same type in the order of their end times. *SLR* maintains the sorted list of events of each type ordered by their end times whose maintenance usually only requiring the assertion of events by the *asserta Prolog* built-in predicate. In this way, finding a previous/next event of a type occurring before/after a time point requires examining only a part of the history of those events.

$$prev(p_n(t_1,...t_n)^{[z_1,z_2]}, z_s) \text{:-} p_n(t_1,...t_n)^{[z_1,z_2]}, z_2 \leq z_s, \\ \neg(p_n(t_1",...t_n")^{[z_1",z_2"]},\ z_2" \leq z_s, z_2" > z_2). \quad (1)$$

$$next(p_n(t_1,...t_n)^{[z_1,z_2]}, z_s, z_e) \text{:-} p_n(t_1,...t_n)^{[z_1,z_2]},\ z_s \leq z_2 \leq z_e, \\ \neg(p_n(t_1",...t_n")^{[z_1",z_2"]},\ z_s \leq z_2" \leq z_e, z_2" < z_2). \quad (2)$$

5.1.3 State-Based Knowledge Representation

SLR aims at simplifying the transformation of events into a state-based representation of knowledge, using derived facts. The following paragraphs presents some typical cases where a state-based representation is more suitable and how it is realized in *SLR*.

Persistent Knowledge Persistent knowledge refers to information that is assumed not to change over time.

Example 2. The following rule specifies that the color of an object at a time T is the color that the object was perceived to have at its last observation.

$$color(O, C)^T \text{:- } prev(obj(O, , C)^Z, T). \tag{3}$$

Persistence with Temporal Validity The temporal validity of persistence refers to the period when it is assumed that information derived from an observation remains valid.

Example 3. To pick up an object O, its location should be determined and sent to a planner to produce a trajectory for the manipulator to perform the action. This task can be naively presented as the sequence of actions: determine the object's location L, compute a manipulation trajectory *Trj*, and perform the manipulation. However, due to environment dynamics and interleaving in task execution, the robot needs to check that the object's location has not been changed and the computed trajectory is still valid before executing the actual manipulation task. The following three rules can be used to determine the location of an object and its validity as follows. If the last observation of the object is within the last five seconds, the object location is set to the location at which the object was seen last time. If the last observation was made longer than five seconds ago, the second rule specifies that the location is outdated. The third rule sets the location to "never-observed", if the robot has never observed such an object. The symbol ! represents *Prolog* cut operator and locations are assumed to be absolute.

$$location(O, L)^T \text{:- } prev(seg(O, L)^Z, T), T - Z \leq 5, !. \tag{4}$$

$$location(O, \text{“}outdated\text{”})^T \text{:- } prev(seg(O, L)^Z, T),\ T - Z > 5, !. \tag{5}$$

$$location(O, \text{“}never\text{-}observed\text{”})^T. \tag{6}$$

Continuous Knowledge Continuous knowledge refers to information from a continuous domain.

Example 4. The following rule calculates the camera to base relative position L at a time T. It interpolates from the last observation L_1 before T to the first observation L_2 after T. *est* is a user defined term performing the actual interpolation.

$$\begin{aligned} tf(cam, base, L)^T \text{:- } & prev(tf(cam, base, L_1)^{T_1}, T), \\ & next(tf(cam, base, L_2)^{T_2}, [T, \infty]), est([L, T], [L_1, T_1], [L_2, T_2]). \end{aligned} \tag{7}$$

The following rule similarly interpolates the base to world relative position L at a time T. However, if the position is not observed within a second after time T, the position is assumed without change and is set to its last observed value. The $\rightarrow$ symbol represents *Prolog* "If-Then-Else" choice operator.

$$tf(base, rcf, L)^T \text{ :- } prev(tf(base, rcf, L_1)^{T_1}, T),$$
$$(next(tf(base, rcf, L_2)^{T_2}, T, T+1) \rightarrow est([L, T], [L_1, L_2], [L_2, T_2]) \ ; \ L \ is \ L_1). \quad (8)$$

The following *ELE* rule concerns recognition of an object O at a position L_{o-c} relative to the camera at a time T. It generates a corresponding *segR* event. It calculates the object position in the reference coordination frame by querying the *SLR* knowledge base. The camera to base and base to world relative positions at time T are estimated by rules (7) and (8).

$$segR(O, L) \leftarrow seg(O, L_{o-c})^T \ WHERE(\ tf(cam, base, L_{c-b})^T,$$
$$tf(base, rcf, L_{b-rcf})^T,$$
$$mul([L_{o-c}, L_{c-b}, L_{b-rcf}], L) \ ). \quad (9)$$

5.1.4 Active Memory

SLR supports selective recording and maintenance of data in knowledge bases using memory instances.

Definition 21 (Memory Instance). A memory instance with an id Id, a query Q and a policy $\langle L, N \rangle$ keeps the record of a subset of input events to *SLR*: the events that match the query Q such that at each time T, the memory instance only contains the events which have their end times within the last L seconds and only includes the recent N number of such events ordered by their end time. An id is a ground term and a query is of the form $\langle e, Cond \rangle$, where e is an event atom and $Cond$ is a set of conditions on variables that are arguments of e. An event P matches a query pattern Q when there is a substitution that can unify p and e and makes the conditions in $Cond$ true (i.e. $\exists\theta(p = q_\theta)$).

Memory instances are created by executing queries of the form *c_mem(Id,Q,N,L)* on the *SLR* knowledge base in initialization of the *SLR* program. They can also be created at runtime by *ELE* rules or by external components using a *ROS* service the *IEC* provides. Similarly, memory instances are deleted at runtime by executing queries of the form $d\_mem(Id)$ each deleting all memory instances whose id_i match the term Id (i.e. $\exists\theta(id = Id_\theta)$).

Example 5. The $c\_mem(tf, \langle tf(X,Y,Z), \langle\rangle\rangle, \infty, 300)$ query creates a memory instance to keep the history of *tf(X,Y,Z)*^{T} events from the *stateRec* component for 300 seconds. In the rule (9), we saw that the *SLR* knowledge base is queried to position object segments in the reference coordination frame. If we assume that the *IEC* receives data of object segments within 300 seconds since they appear in front of the camera, then we only need to keep the history of *tf* events for 300 seconds. In another example, for each object o_i in *segR(O,L)* events, the *ELE* rule (10) generates a memory instance with the corresponding id of $obj(o_i)$. A memory instance is generated, if it does not already exist. This is checked using the $\neg exist\_mem(monitor(O))$ clause. Each memory instance $obj(o_i)$ keeps the last occurrence of $segR(o_i, L)$ events at which o_i is located on the floor, checked by the *onFloor Prolog* term implementing the required spatial inference. The use of *DO* clause is another way of performing *SLR* queries in *ELE* syntax.

$$Do(c\_mem(obj(O), \langle segR(X,L), \langle X == O, onFloor(L)\rangle\rangle, 1, \infty)) \leftarrow segR(O,L)WHERE(\neg exist\_mem(obj(O))). \quad (10)$$

The histories of events maintained in memory instances are accessed in the *SLR* program using the following static terms.

Definition 22 (Memory Term Semantics). A $mem(Id, X)$ term unifies X with an event $p_n(t_1, .., t_{n-2})^{[z1,z2]}$ that belongs to a memory instance whose *id* matches the term Id (i.e. $\exists\theta(id = Id_\theta)$). When backtracking over a $mem(Id, X)$ term in evaluating an *SLR* query, the possible unification of X is checked against all events recorded in all such memory instances.

Definition 23 (Previous_Memory Term Semantics). A term of the form $prev(Id, X, Z_s)$, where Id is a ground term, unifies X with an event which has the latest occurrence time among the events that belong to the memory instance Id, are unifiable with X and have their end time before or equal to Z_s. The term fails if such a unification is not found.

Definition 24 (Next_Memory Term Semantics). A $next(Id, X, z_s, z_e)$ term, where Id is a ground term, unifies X with an event which has the earliest occurrence time among the events that belong to the memory instance Id, are unifiable with X and have their end time within time interval $[Z_s, Z_e]$. The term fails if such a unification is not found.

Example 6. The rule (11) re-writes the rule (8) by querying the previous event of the form *tf(base,rcf,L)* occurring before T and the next *tf(base,rcf,L)* event occurring

during $[T, T+1]$ from the memory instance *tf*, defined in the previous example to keep the history of *tf* events for 300 seconds. Another example is the query $findAll$(X, *mem(obj(O),X)*, *List*) which queries all $obj(O)$ memory instances created by the rule (10) for their records of $segR(X, L)$ events using the $mem(obj(O), X)$ template and put the list of results in the variable *List*.

$$\begin{aligned} tf(base, rcf, L)^T \text{ :- } & prev(tf, tf(base, rcf, L_1)^{T_1}, T), \\ & (next(tf, tf(base, rcf, L_2)^{T_2}, T, T+1) \rightarrow \\ & est([L, T], [L_1, L_2], [L_2, T_2]) \ ; \ L \ is \ L_1). \end{aligned} \tag{11}$$

SLR generates events when memory instances are created, deleted or updated. Memory events are fed to *ELE* as input. Consequently, patterns of memory events can be captured by *ELE* to notify external components with information about changes of memory. Memory events are also used internally to keep track of the latest update time of memory instances. This mechanism is used to synchronized queries, discussed in Section 5.1.5.

This mechanism can be used to generate all sorts of events related to changes of the memory such as the addition or deletion of memory instances or even the addition or deletion of events to/from memory instances.

5.1.5 Synchronizing Queries over Asynchronous Events

SLR supports the synchronization of queries to deal with the delayed and out of order reception of sensory data to the knowledge base.

Definition 25 (Event Process Time). The process time (i.e. $t_p(e)$) of an event e is the time at which the event is received by and added to the *SLR* knowledge base (i.e. processed by *IEC*).

Definition 26 (Event Delay Time). The delay time ($t_d(e)$) of an event e is the difference between its process time and its end time (i.e. $t_d(p^{[z1,z2]}) = t_p(p^{[z1,z2]}) - z2$).

A query should be evaluated after all events relevant to the query have been already received by the *SLR* knowledge base. The parameter z of a query $\langle goal, z \rangle$ limits the scope of the query to observations made up until time z. To evaluate the *goal*, a number of memory instances are queried. Therefore, all relevant events to these memory instances occurring up to time z should have been received by *SLR* before performing the query.

Definition 27 (History Availability). The history of events of a type p_n up to a time z is available at a time t when at this time the *SLR* has received all events of type p_n occurring by time z (having end time earlier or equal to z).

Moreover, all previous and next memory terms should be correctly evaluated according to their definitions. Finding the previous event of type $p_n(t_1, .., t_n)$ occurring up to time z_s requires having received all $p_n(t_1, .., t_n)$ events occurring up until time z_s. If we assume events of each type are received by *SLR* in the order of their end times, then finding the next event of type $p_n(t_1, .., t_n)$ occurring within time interval $[z_s, z_e]$ requires having received the first $p_n(t_1, .., t_n)$ event which has its end time equal or more than z_s, or make sure that no $p_n(t_1, .., t_n)$ event has occurred during $[z_s, z_e]$. *SLR* postpones an individual query[16] when necessary until it is achievable, as defined below.

Definition 28 (Dynamic Goal Set of Query). The dynamic goal set of a query $\langle goal, z\rangle$ for an *SLR* program D is the set of all $mem(Id, X)$, $prev(Id, Z_s)$ and $next(Id, Z_s, Z_e)$ predicates that can possibly be queried when evaluating the *goal* on the knowledge base. The dynamic goal set can be determined by going through all rules in D using which the *goal* could be possibly proven and gathering all $mem(Id, X)$, $prev(Id, Z_s)$ and $next(Id, Z_s, Z_e)$ terms appearing in bodies of those rules.

Definition 29 (Query Achievability). A query $\langle goal, z\rangle$ is achievable when three conditions are met. First, the histories of all relevant events to memory instances in dynamic goal set of the query are available up to time z. Second, for each $prev(Id, Z_s)$ term in the dynamic goal set of the query, the history of all relevant events up to time Z_s is available. Third, for each $next(Id, Z_s, Z_e)$ term in the dynamic goal set of the query, a relevant event has been received or the history of all relevant events up to time z_e is available.

To determine when the history of events of a type p_n up to a time z is available, *SLR* can be programmed in two complementary ways. One way is to set a maximum delay time (i.e. $t_{d_{max}}$) for events of each type. When the system time passes $t_{d_{max}}(p_n)$ seconds after z, *SLR* assumes that the history of events of type p_n up to time z is available. The maximum delay times of events depends on the runtime of the components generating them and need to be approximated. The maximum delay times can be set the system developer. It can also be approximated by *SLR* as follows. Whenever an event of type p_n is processed, *SLR* checks its delay, the

[16] Postponing one query does not delay the others.

difference between its end time and the current system time, and sets the $t_{d_{max}}(p_n)$ to the maximum delay time of p_n events encountered so far. When smaller maximum delay times of events are assumed, queries are evaluated sooner and hence the overall system works in more real-time fashion, but there is more chance of answering a query when the complete history of events asked by the query is not in place yet. When larger maximum delay times of events are assumed, there is a higher chance to have all sensory data up to the time specified by the query already processed by the corresponding components and their results received by *SLR* when the query is evaluated. However, queries are performed with more delays.

The other way that *SLR* can ensure to have received the full history of events of a type p_n up to a time z in its knowledge base is by being told so by a component generating such events using special $updated(p_n)^z$ events. Whenever *SLR* receives such an event, it assumes that the history of events of the type p_n up to time z is available.

The query synchronization is often required for a query that interpolates the value of an attribute at a given time using *next* and *prev* term. The value can be interpolated as soon as the first relevant event after that time is received. *SLR* monitors memory events, discussed in Section 5.1.4, and evaluates the postponed queries as soon as necessary events are received.

Example 7. When the position of an object O in the world coordination frame at a time T is queried by the rule (9), the query can be answered as soon as both camera to base and base to world relative positions at time T can be evaluated by rules (7) and (8). The former can be evaluated (i.e. interpolated) as soon as *SLR* receives the first $tf(\text{`}cam\text{'}, \text{`}base\text{'}, P)$ event with a start time equal or later than T. The latter can be evaluated as soon as the *SLR* receives the first $tf(\text{`}base\text{'}, \text{`}world\text{'}, P)$ event with the start time equal or later than T, or when it can ensure that no $tf(\text{`}base\text{'}, \text{`}world\text{'}, P)$ event has occurred within $[T, T+1]$. If we assume $t_{d_{max}}(tf(\text{`}base\text{'}, \text{`}world\text{'}, P))$ is set to 0.5 second, *SLR* has to wait 1.5 second after T to ensure this.

Example 8. The robot is asked about the objects it sees on *table1*. To answer the question, the robot takes a number of pictures from the table starting at time t_1 and finishing by time t_2 and then the *SLR* knowledge base is queried by $\langle goal, t_2 \rangle$ where the *goal* is

$$\begin{aligned} &\textit{findall}(obj(O, Type, L), \\ &\quad (\ mem(obj(O), segR(O, L)^{T_x}), \ t_1 \leq T_x \leq t_2, prev(obj(O, Type, P)^{T_y}, t_2) \ ), \\ &\quad List) \end{aligned} \tag{12}$$

The query result is the list *List* of terms of the form *obj(O,Type,L)* matching the template specified by the second argument of the *findall* term. This includes all object segments recorded as $segR(O,L)^{T_x}$ events in $obj(O)$ memory instances recognized during $[t_1, t_2]$. The type of each object segment o_i is recognized by querying the last $obj(o_i,Type,P)$ event occurring before or at time t_2. To list all the objects, *SLR* makes sure to evaluate the query after the histories of both *segR(O,L)* and *obj(O,Type,L)* events up to time t_2 are available. A signaling mechanism to realize this is as follows. After finishing the processing of each image taken at a time t and outputting the recognized object segments, the *segRec* component sends out the event $updated(segR)^t$. The *IEC* receives these events sending object segments whose type is not known and the $updated(segR)^t$ events to the *objRec* component. We assume events of each type are communicated among the components in order. The *objRec* component receives some object segments recognized at a time t, processes them in the order it receives them and sends the recognized types back to the *IEC*. Whenever the *objRec* processes an $updated(segR)^t$ event, it realizes that it has finished processing of the object segments recognized up to time t and generates an $updated(obj)^t$ event. Receiving $updated(segR)^t$ and $updated(obj)^t$ events, *SLR* is notified when the histories of both types of events up to time t_2 are available and then evaluates the query.

5.2 Discussion

The use of memory in existing research includes collecting data from various sources and in time, mediating as a shared resource for component interaction (i.e. blackboard architectural pattern [77]), refining data by various processes, and integrating various reasoning capabilities to maintain and query the robot's knowledge of the environment for task execution, human interaction and learning [13, 80, 65, 37, 36, 71, 72, 50, 49, 52, 56]. A large set of on-demand information processing requirements have been discussed elsewhere [49, 80].

A main concern in supporting on-demand information processing is the choice of language for representing and storing data. The choice of language and its execution system largely determines the extent to which various on-demand information processing requirements along data, process, memory and access dimensions are

supported, perhaps the most important ones being knowledge modelling and reasoning. The advantage of non logic-based data representations, for example, using programming data structures in *CAST* [36, 71] and *GSM* [56], is the flexibility and efficiency in the representation and manipulation of amodal data such as image data and probability distributions. However the expressiveness of queries for information maintained by such systems is limited. An interesting approach is the XML data representation by *IDA* [80, 65] supporting *Xpath* queries [15], for example, to retrieve data of objects recognized with confidence of more than a threshold. The data representation in non-logic based systems is usually tightly related to the data representation used in their underlying framework and does not support logical reasoning. In *Retalis*, binary data is represented as *String*. This requires encoding binary data to the *Prolog String* format when importing a *ROS* message to *Retalis* and decoding it when the data is sent back to *ROS*, which is time consuming. However, one can maintain the actual binary objects in *c++* and manipulate handlers to the objects in *Retalis*.

A recent survey of existing robotic *information management* systems [49] shows that most systems rely on logical formalisms, mainly including declarative languages such as the *OWL*[17] language [57] based on Description logics [10] and/or rule-based languages such as the *SWRL*[18] language [45] for rule-based reasoning in *OWL* and *Prolog*. In particular, *OWL* is a popular choice to define ontologies of various types of knowledge such as knowledge of space, objects, actions and robot capabilities used, for instance, in *ORO* [50, 49], *KnowRob* [72, 71] and *OUR-K* [52]. Defining ontologies are necessary to integrate various sources of knowledge such as the domain and common sense knowledge as performed by the aforementioned systems and for sharing robots' knowledge, for instance, in the cloud [73]. While we did not address modeling of knowledge, existing ontologies can be directly used in *Retalis* as *OWL* ontologies can be represented and reasoned upon in *Prolog*. For example, *KnowRob* offers one of the most comprehensive robotic ontologies and uses the *Prolog Semantic Web Library*[19] [60] for loading and storing *RDF*[20] [21] triples and the *Thea*[21] *OWL* parser library [75] for OWL reasoning on top of this representation.

The use of *Prolog* as the underlying technology for maintaining robotic *OWL* knowledge has a few practical advantages for inference compared to the use of existing description logic reasoners such as the *Pellet*[22] reasoner [68] used in *ORO*.

[17] http://www.w3.org/TR/2004/REC-owl-ref-20040210/
[18] http://www.w3.org/Submission/SWRL/
[19] http://www.swi-prolog.org/pldoc/package/semweb.html
[20] http://www.w3.org/RDF/
[21] http://www.semanticweb.gr/thea/
[22] http://clarkparsia.com/pellet/

Those reasoners keep a classified version of the knowledge base in memory specifying each individual belonging to which classes. Therefore continuous changes of the knowledge base through acquiring sensory data requires frequent re-classification of the whole knowledge which can be costly [72]. This problem can be partially addressed by optimizing this operation using an incremental updating technique [34]. The more important advantage is related to the open world assumption in *Description Logics* versus the closed world assumption in *Prolog*, and the monotonicity of description logics versus supporting a form of non-monotonicity in *Prolog* by the *negation as failure* inference rule within the closed world assumption. In the closed world assumption, representations can be more compact as 'a fact not being true' does not need to be described but it can be inferred by not being able to prove the fact. Moreover, the open world assumption and monotonicity of *Description Logic* makes the representation and reasoning on dynamics of the environment (i.e. changes and actions) difficult requiring to handle such aspects externally [84, 49], but, for instance, *KnowRob* implements a predicate to return an object's location at a time by searching for the last observation of the object's location before that time. Reasoning about changes and actions has been extensively studied in various knowledge formalisms such as Situation Calculus [51] and *Event Calculus* [47, 67]. The *SLR* language provides a practical and efficient solution for representing robot knowledge based on discrete observations, providing a means to deal with the temporal validity of data and representation of continuous domains which is not the focus of such formalisms. Compared to the *KnowRob* approach of, for instance, implementing a predicate to represent an object's location at a time, *SLR* simplifies the definition of such predicates in general and increases the efficiency of their computations by maintaining the sorted list of events based on their occurrence times. *Prolog* provides a flexible support for access to external data or reasoning functionalities while reasoning on knowledge through procedural attachments to the *Prolog* terms. This feature is used in *KnowRob*, for instance, to compute spatial relations between objects and in *Retalis* to integrate *OpenGL Mathematics*[23] (GLM) for arithmetic operations.

To the best of our knowledge the *SLR* support for synchronization of queries on knowledge built upon asynchronous data is not presented elsewhere. However, similar synchronization mechanisms as found in *SLR* are implemented in other robotic software in a more limited context. One example is the *DyKnow* framework [38] that synchronizes data received from streams of data based on different policies to generate new ones. Another example is the *tf* library [32] widely used in *ROS* for querying position transformation between robot's coordination frames over time.

[23] http://glm.g-truc.net/0.9.5/index.html

When a relative position at a time is queried, the query is not answered until receiving the first observation of that position at or after that time. The *tf* library only supports interpolation of data similar to the *SLR* rule (7). Therefore, even if a position is constant in time, its value needs to be continuously published to *ROS* consuming the network bandwidth. Moreover, sometimes a component such as *AMCL* in *ROS* provides updates in a slow rate but they are precise enough to be used until the next update is made available. In order to not delay the processing of data until availability of the next update, this component stamps its updates in the future.[24] Apart from being semantically confusing, time stamping updates in future can result in using old data even if new data is already available. With the *SLR* extrapolation approach, for instance, implemented by the rule (8), if a position transformation is static, its value does not need to be published being extrapolated from its last observed value. In addition, the time bound of the *next* predicate in *SLR* allows to specify how long *SLR* needs to wait to see whether a value has been changed, assuming after each relevant change a notification is received.

Except a few, most information management systems leave pruning data from the memory to external components. In *ORO*, knowledge is stored in different memory profiles, each keeping data for a certain period of time. In *IDA*, scripts are activated periodically or in response to events of memory changes to perform garbage collection. In *SLR*, flexible garbage collection functionalities are blended in the syntax of the language. In addition, a subtle difference between *SLR* and other systems is that in the existing systems, external components store the data in memory. In *SLR*, memory instances are declaratively defined which selectively store data from the input flow of events to the *SLR*. The storage of data in *SLR* is similar to active memories such as the ones of *IDA* and *CAST* as data is recorded in memory instances with unique identifiers, however *SLR* supports logical reasoning over the contents of memory instances. This approach supports having different memory profiles for different pieces of data and a flexible way of selecting the data that are to be reasoned about as a whole, thus allowing to reason about a part of knowledge that could be inconsistent with other part of the knowledge maintained in the memory. Furthermore, active memories allow external components to update the contents of memory instances. As such, suitable error handling and locking mechanisms are necessary to synchronize the parallel access to memory. In contrast, the modeling of the input as a stream of events and clear semantics of memory instances in *SLR* removes much of the problems related to the parallel access of data. For an example, consider two components processing object segment events to recognize the orientation and type of objects. In our approach, this can be

[24] http://wiki.ros.org/tf/FAQ

implemented as follows: an object segment event is sent to both components, these components perform their processes and generate their uniquely typed events. Then an *ELE* rule receives events from these components, synchronizes them based on their object identifiers and occurrence times and produces new events of recognized objects with their types and orientations. In a naive approach, object segments are recorded in the memory and are processed and updated by both components in parallel which could re-write each other results.

SLR supports notifying external components when memory instances are added or deleted to the memory. This can be easily extended to also generate corresponding notifications when events are added or deleted from memory instances. However, the input flow of events to *SLR* is processed by *ELE*. Therefore external components can subscribe to *Retalis* to be notified when the data of interest is being fed to *SLR*. While notifying changes of the memory is a main functionality in active memories, it is less common in logic-based knowledge management systems. An exception is *ORO* to which one can subscribe to receive a notification, whenever a fact can be inferred by the *ORO* knowledge base. However it is not described whether or not this includes the knowledge that can be derived by *SWRL* rules. Moreover, it not described whether this functionality is implemented by continuously querying the knowledge base for such a fact, or it is efficiently realized by an incremental and event-driven algorithm such as backward chaining rules in *ELE* [6, 5, 3].

6 Evaluation

This section evaluates the performance of *Retalis* by demonstrating the implementation of an application for a NAO robot. NAO is a small programmable humanoid robot offered by Aldebaran Robotics[25], equipped with advanced sensors such as cameras, touch sensors and microphones. In the application, NAO observes objects in the environment, perceiving their relative positions to its camera, and computes the position of objects in the environment. Figure 5 presents software components[26] of the NAO application, operating as follows. The *NAO nodes*[27] component provides an interface to acquire sensory data and to command the NAO robot. It publishes images generated by the top camera of the robot. It also publishes events about the transformation among the robot's coordinate frames. Each of these events contains a set of transformations where each transformation specifies the relative position

[25] http://www.aldebaran.com/en

[26] The software includes also a face recognition component which is not discussed for brevity.

[27] http://wiki.ros.org/nao_robot

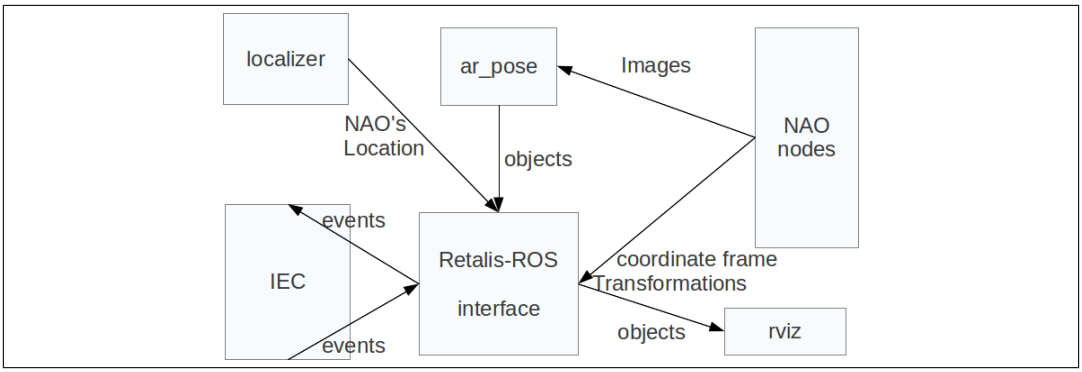

Figure 5: NAO's software components

among two coordinate frames. The *ar_pose*[28] component processes the images to recognize objects and calculates the position of objects with respect to the camera. Each event from *ar_pose* contains data of a set of observed objects. The *localizer* component calculates the robot's position in the world. The *IEC* component is subscribed to information about objects' positions, robot's location and coordinate transformations. It calculates the position of objects in the world from the transformation among the following pairs of coordinate frames, *(world, base_link), (base_link, torso), (torso, neck), (neck, camera) and (camera, object)*. The arithmetic operations are performed using the *OpenGL Mathematics*[29] (GLM) library which has been integrated in *Retalis*. The *rviz*[30] component visualizes the objects in the environment. The *IEC* communication with other nodes is realized by the *Retalis-ROS interface* component. This component converts *ROS* messages to *Retalis* events and vice versa. The *IEC* and the *Retalis-ROS interface* components are implemented in *Retalis*.

6.1 Basic setup

For a first test implementation, all software components run remotely on an XPS Intel Core i7 CPU@ 2.1 GHz x 4 laptop running ubuntu 12.04 LTS, connected to the NAO robot. After the evaluation phase, the software will be implemented in the NAO robot itself. NAO comes with an Intel Atom CPU@1.6 GHz running Linux. The performance is evaluated by measuring the CPU time, the amount of time of a CPU of the computer that is used by the *Retalis* program. We measure the CPU time

[28] http://wiki.ros.org/ar_pose
[29] http://glm.g-truc.net/0.9.5/index.html
[30] http://wiki.ros.org/rviz

as the percentage of the CPU's capacity (i.e. CPU usage percentage) computed by the operating system. In the following graphs, the vertical axis represents the CPU usage percentage and the horizontal axis represents the running time in seconds. The CPU time is logged every second and is plotted using "gnuplot smooth bezier".

The NAO application includes the following tasks:

- On-flow processing: events from *ar_pose* and *NAO nodes* are split into respective events such that each event contains data of a single object or the transformation among a single pair of coordinate frames. The transformation data among pairs of coordinate frames are published with frequencies from 8 to 50 hertz. There are in average 7 objects perceived per second. In total, *Retalis* processes about 1900 events per second.

- Memorizing and forgetting: there are 5 memory instances observing the events. They record and maintain the last 30 seconds histories of the transformation among the pairs of coordination frames used to calculate the transformation among *world* and *camera*.

- Querying memory instances: for each observed object, *SLR* is queried for the *world-to-camera* transformation. The transformation among a pair of coordinate frames at a time is calculated by interpolation, as performed by the rule (11). Each interpolation requires accessing a memory instance twice, once using a *prev* term and once using a *next* term. To calculate the position of all objects, memory instances are accessed 70 times per second.

- Synchronization: a query is delayed in case any of the necessary transformations can not be interpolated from the data received so far. *Retalis* monitors the incoming events and performs the delayed queries as soon as all data necessary for their evaluations are available.

- Subscription: there are 8 distinct objects in the environment and consequently 8 subscriptions to publish recognized objects to distinct *ROS* topics. The *rviz* component is subscribed to these topics to visualize the position of objects.

Figure 6 shows the CPU time used by the *Retalis* and *Retalis-ROS*-converter nodes when running the NAO application. The *Retalis* node calculates the position of objects in real-time. It processes about 1900 events, memorizes 130 new events and prunes 130 outdated events per second. It also queries memory instances, 70 times per second. These tasks are performed using about 18 percent of the CPU time. In this experience, the *Retalis* node has been directly subscribed to *ROS* messages containing information about coordinate transformations and recognized

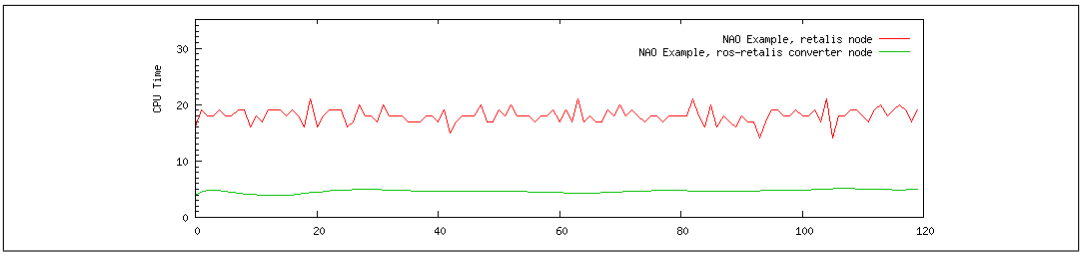

Figure 6: NAO application

objects. The *Retalis-ROS*-converter, consuming about 5 percent of CPU time, only subscribes *Retalis* to the recognized faces and converts and publishes events about objects' positions to *ROS* topics.

As we saw in Section 3, *Retalis* provides an easy way to subscribe to *ROS* topics and automatically convert *ROS* messages to events. This is implemented by the *Retalis-ROS*-converter node. The implementation is in Python and is realized by inspecting classes and objects at runtime and therefore is expensive. Figure 7 shows the CPU time used by the *Retalis* and *Retalis-ROS*-converter nodes for the NAO application, when the *Retalis-ROS*-converter is used to convert all *ROS* messages to *Retalis* events. In the previous configuration, the conversion from *ROS* messages, containing information about coordinate transformations and recognized objects, to events was performed by a manually written *c++* code, rather than using the *Retalis* automatic conversion functionality written in *Python*. We observe that in the new configuration, the *Retalis* node consumes a few percent less, but the *Retalis-ROS*-converter node consumes about forty percent more CPU time, comparing to the previous configuration. These results show that while the automatic conversion among messages and events are desirable in a prototyping phase, the final application should implement it in C++ for performance reasons. We will investigate the possibility to optimize and re-implement the *Retalis-ROS*-converter node in C++.

Metric evaluation of languages and systems like *Retalis*, in general, is challenging for the following reasons[49, 48, 56]. Experiments often involve many other modules running in parallel and building repeatable experiments for robots in dynamic environments is challenging. In addition, very few existing systems report metric evaluations and the lack of standard *API*s and differences in functionalities makes it hard to compare these systems. The rest of this section evaluates main *Retalis* functionalities. We report a number of experiments using data from the NAO application, recorded by *rosbag*.[31] Using *rosbag*, data can be played in a simulation, as if it is played in real-time. While single performance results in the following ex-

[31] http://wiki.ros.org/rosbag

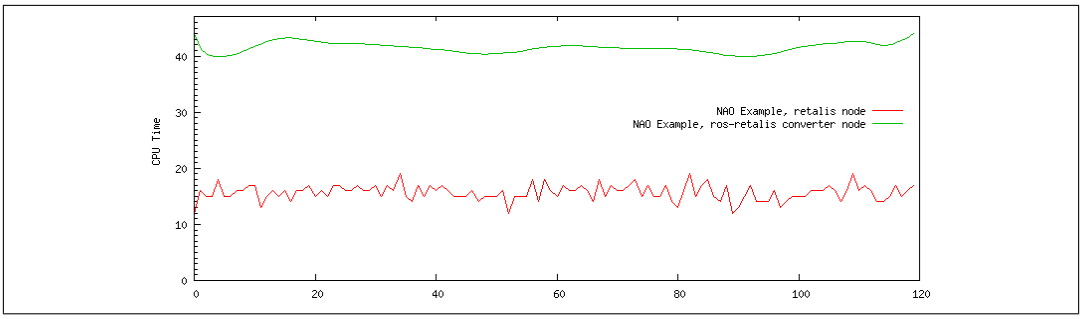

Figure 7: NAO application with automatic conversion of messages and events

periments depend on the NAO application, a series of experiments is presented for each functionality, allowing us to make a number of general observations about the performance of *Retalis* functionalities.

6.2 Forgetting and Memorizing

This section evaluates the performance of the memorizing and forgetting functionalities. We measure the CPU time for various runs of the NAO application where the numbers and types of memory instances are varied. We discuss the performance of memory instances by comparing the CPU time usages in different runs.

When an event is processed, updating memory instances includes the following costs:

- Unification: finding which memory instances match the event.
- Assertion: asserting the event in the database for each matched memory instance.
- Retraction: retracting old events from memory instances that reached their size limit.

Figure 8 shows the CPU time for a number of runs where up to 160 memory instances are added to the NAO application. These memory instances record *a(X,Y,Z,W)* events. Among the events processed by *Retalis*, there are no such events. The results show that the increase in CPU time is negligible. This shows that a memory instance consumes CPU time only if the input stream of events contains events whose type matches the type of events the memory instance records.

In Figure 9, the green and blue lines show the CPU time for cases where 20 memory instances of type tf(X,Y,V,Q) are added to the NAO application. These

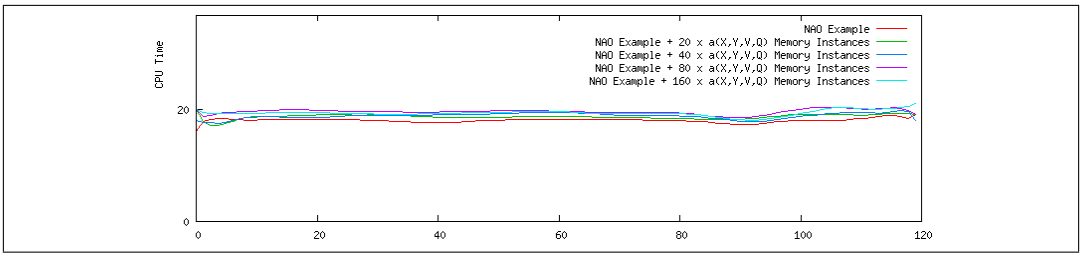

Figure 8: Irrelevant memory instances

memory instances match all tf events, about 1900 of such is processed every second. The size of memory instances for the green line is 2500. These memory instances reach their size limit in two seconds. After this time, the CPU time usage is constant over time and includes the costs of unification, assertion and retraction for updating 20 memory instances with 1900 events per second. The size of memory instances for the blue line is 150,000. It takes about 80 seconds for this memory instances to reach their size limit. Consequently, the CPU time before the time 80 only includes the costs of unification and assertion, but not the costs of retraction. After the time 100, the CPU usages of both runs are equal. This shows that the cost of a memory instance does not depend on its size.

The purple line shows the CPU time for the case where similarly there are 20 memory instances of type tf(X,Y,V,Q). However, these memory instances record events until they reach their size limit. We added a condition for these memory instances such that after reaching their size limit, they perform no operation when receiving new events. After the time 100, the CPU time is constant about 23 percent, being 5 percent more than the CPU time of the NAO application, represented by the red line. This 5 percent increase represents the unification cost. This also shows that the costs of about 38000 assertions and 38000 retractions per second is about 30 percent of CPU time. In other words, 2500 memory updates (i.e. assertions or retractions) are processed using one percent of CPU time.

Figure 10 shows the CPU time for a number of runs where up to 40 memory instances of type tf(X,Y,Z,W) and size 2500 are added to the NAO application. The red line at the bottom shows the CPU time for the NAO application. We make the following observations. Adding first 10 memory instances to the NAO application increases the CPU time about 20 percent. After that, adding each set of 10 memory instances increases the CPU time about 13 percents. This shows that the cost grows less than linearly. The implementation of memory instances is in a way that the cost of an assertion or a retraction can be assumed constant. This means that the unification cost for the first set of memory instances is the highest. In other words,

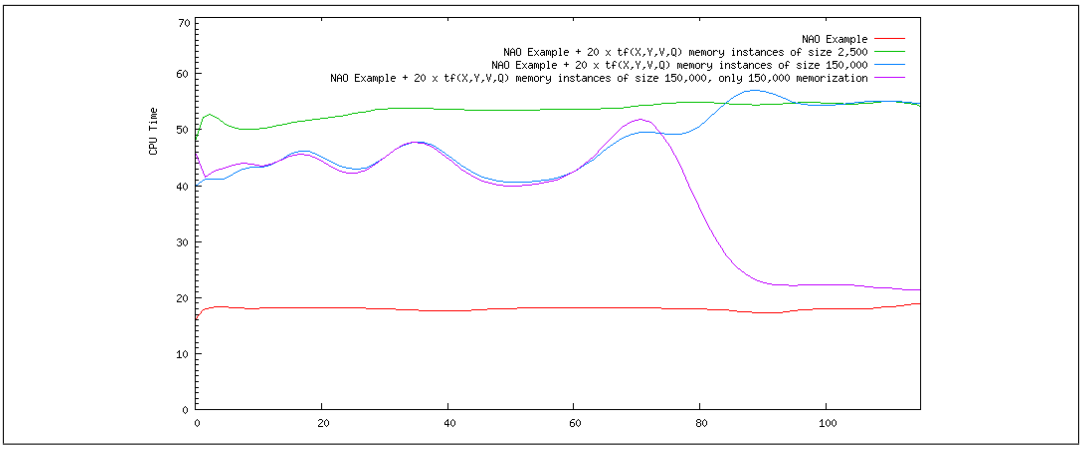

Figure 9: tf(X,Y,V,Q) memory instances (1)

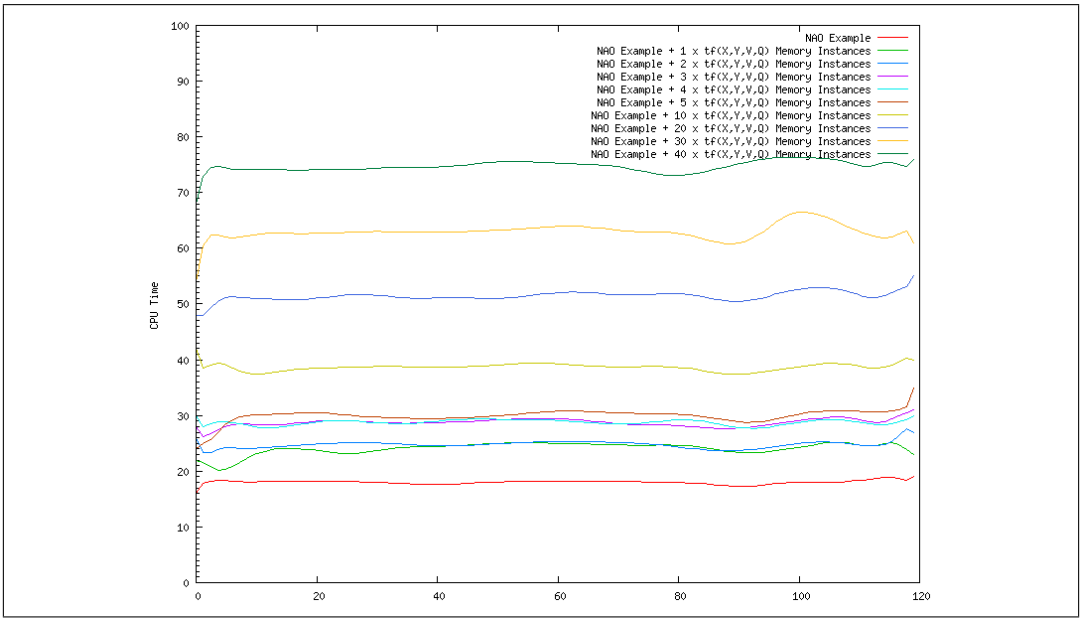

Figure 10: tf(X,Y,V,Q) memory instances (2)

the unification cost per memory instance decreases when the number of memory instances are increased. The reason relates to the way that the underlying *SWI-Prolog* engine searches and unifies terms which is not investigated here.

Figure 11 shows the CPU time for a number of runs where up to 640 memory instances of type tf(head,camera,Z,W) and size 2500 are added to the NAO applica-

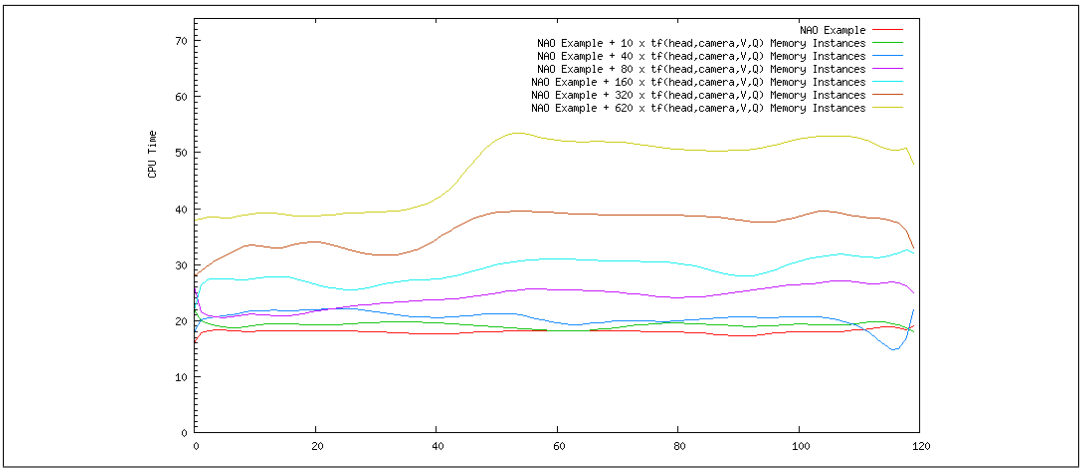

Figure 11: tf(head,cam,V,Q) memory instances

tion. The events matching these memory instances are received with the frequency of 50 Hz. We make the following observations. First, it takes 50 seconds for these memory instances to reach their size limit. After 50 seconds, these memory instances reach their maximum CPU usages, as the costs of retraction is added. Second, each memory instance filters 1900 events per second recording about two percent of them. The cost of 640 memory instances is about 35 percent of CPU time. Third, the unification cost per memory instance is decreased when the number of memory instances are increased.

Figure 12 compares the costs of different types of memory instances. The purple line shows the CPU time for the case where there are 10 memory instances of type tf(X,Y,V,Q). The green line shows the CPU time for the case where there are 320 memory instances of type tf(head,cam,V,Q). We observe that the costs of both cases are equal. The memory instances in the former case record 19,000 events per second (i.e. 10*1900). The memory instances in the latter case filter 1900 events per seconds for tf(head,cam,V,Q) events, recording 16000 events per second (i.e. 320*50). The results show the efficiency of the filtering mechanism.

The brown line shows the CPU time for the case where there are 10 memory instances of type tf(X,Y,V,Q) and 320 memory instances of type tf(head,cam,V,Q). Comparing it with the green and purple lines shows that the CPU time usage of these memory instances is less than sum of the CPU usages by 10 tf(X,Y,V,Q) memory instances and 320 tf(head,cam,V,Q) memory instances. This shows that the unification cost per memory instance is decreased when the number of memory instances are increased, even when the memory instances are not of the same type.

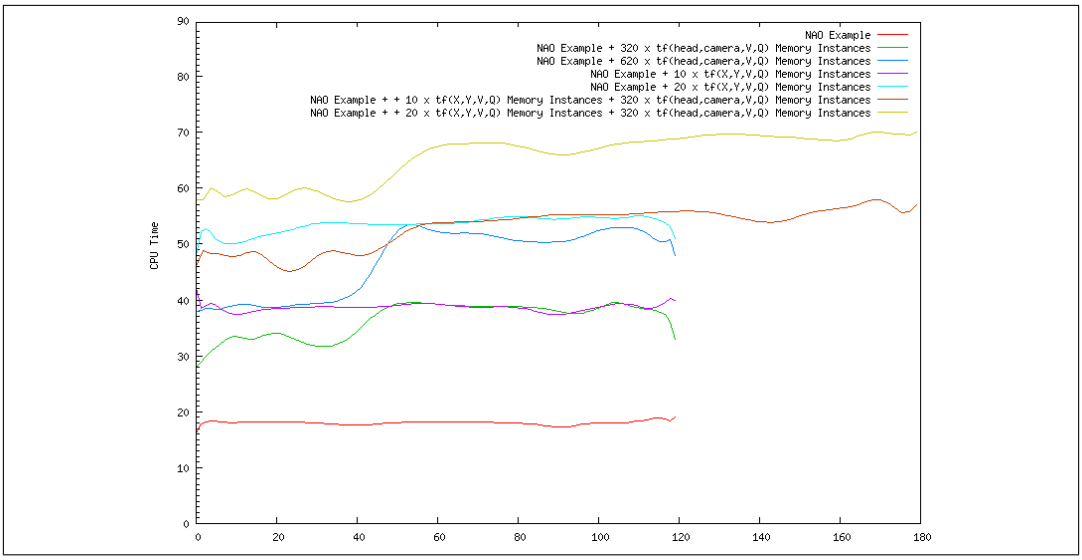

Figure 12: Memory instances of different types

These experiments show that *Retalis* is able to maintain a history of a large volume of data. Memorizing and forgetting functionalities of *SLR* have been optimized as follows. A memory instance memorizes an event by creating an event record containing the event and the identifier of the memory instance. The event record is asserted as the top fact in the database. This operation takes a constant time. Event records of a memory instance are numbered in order of the event occurrence times. *SLR* generates a hash key for each event record, based on the respective identifier and the record number. Event records are indexed on their hash keys. Consequently, accessing an event record takes a constant time *SLR* keeps track of the number of the oldest event record of each memory instance. Therefore, forgetting takes a constant time, irrelevant of the size of memory instances.

6.3 Querying

Retalis queries are *Prolog*-like queries executed by the *SWI-Prolog* system. The following evaluates the performance of next and prev terms and the synchronization mechanism which are specific to *Retalis*. The performance of next and prev terms are important because the sensory data recorded by *Retalis* is queries using these terms. Not only does *Retalis* extend the *Prolog* language with these built-in terms to provide easier syntax for querying history of data, but also to make querying of data more efficient.

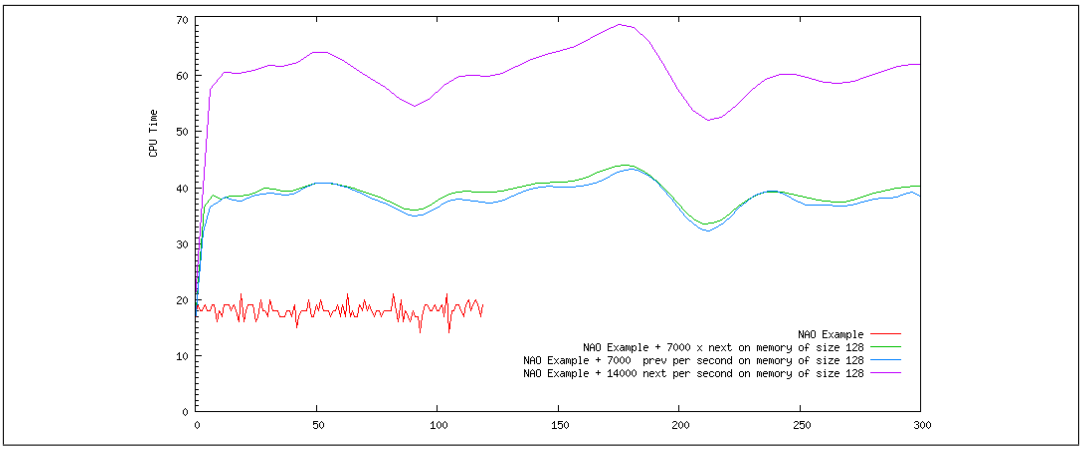

Figure 13: Next and prev terms (1)

Querying Memory Instances

This section evaluates the performance of prev and next terms used to access event records in memory instances. *Retalis* optimizes the evaluation of these terms as follows. It keeps track of the number of event records in each memory instance. The prev and next terms are evaluated by a binary search on event records. An access to an event record by its number takes a constant time. Consequently, the evaluation of prev and next is done in logarithmic time on the size of the respective memory instance. In Figures 13, 14 and 15 below, the red line visualizes the CPU time of the NAO application.

The green line in Figure 13 visualizes the CPU time of the NAO application adapted as follows. There is an additional tf(head,cam,V,Q) memory instance of size 128. This memory instance is queried by 1000 next terms for each recognition of an object. In average, 7000 next terms are evaluated per second. The blue line visualize the CPU time of a similar program in which 7000 prev terms are evaluated per seconds. The figure shows that the costs of the evaluations of prev and next terms are similar. The purple line shows the CPU time of the case where 14,000 next terms are evaluated per second. We observe that the cost grows linearly.

The blue line in Figure 14 visualizes the CPU time of the case where 7000 next terms are evaluated per second. The green line visualizes the CPU time of the case where there are 320 tf(head,cam,V,Q) memory instances added to the NAO application. The purple line visualizes the CPU time of the case where 7000 next terms are evaluated per second and there are 320 tf(head,cam,V,Q) memory instances. We observe that the cost of accessing a memory instance does not depend on existence

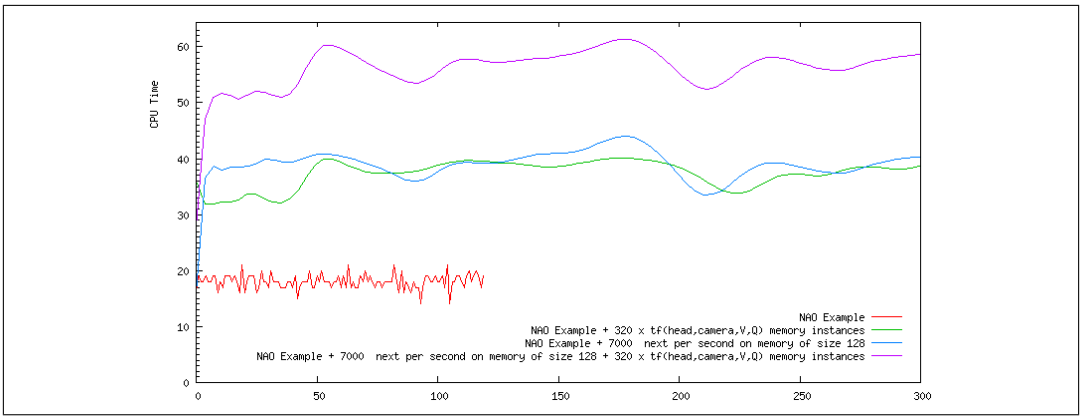

Figure 14: Next and prev terms (2)

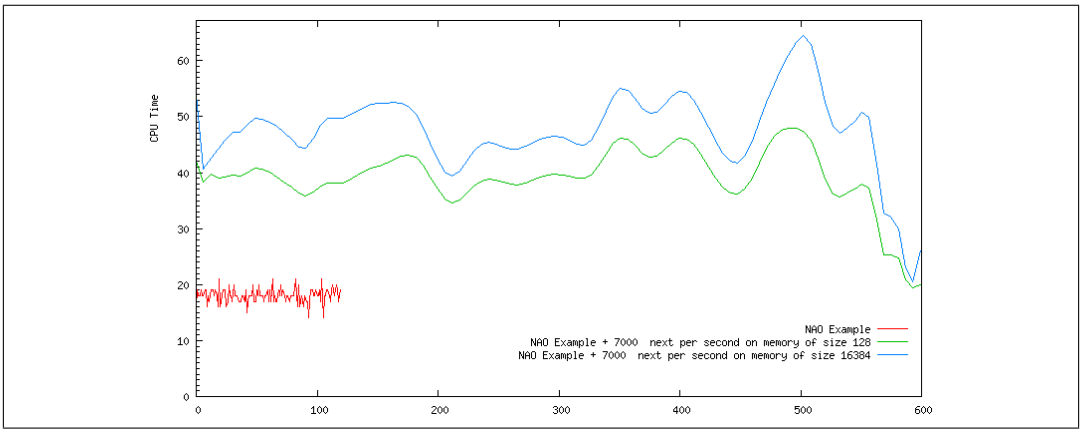

Figure 15: Next and prev terms (3)

of other memory instances.

The green line in Figure 15 visualizes the CPU time of evaluating 7000 next terms per second on a memory instance of size 128. The blue linevisualizes the CPU time of evaluating 7000 next terms per second on a memory instance of size 16384. The size of the memory instance in the latter case is the power of two of the size of the memory instance in the former case. The increase in the CPU time for the latter case, with respect to the NAO application, is less than two times of the increase in the CPU time for the former case.

The *prev* and *next* terms provide efficient ways of accessing records of events. Otherwise, all event records should be read, for instance, to find the latest position of an object. For example, an experiment is reported for the *KnowRob* knowledge

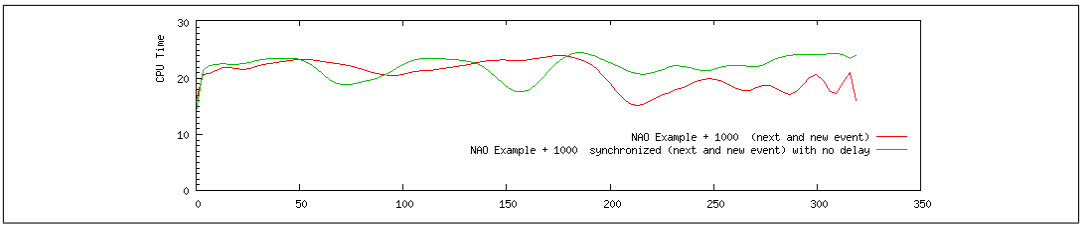

Figure 16: Synchronization with no delay

base where there are 65,000 records of events about the location of an object. It takes 11 seconds to find the latest location [72].

Synchronization

The synchronization mechanism is implemented as follows. Before evaluating a query, memory instances are checked whether they are up-to-date with respect to the query (i.e. the query is achievable as defined in Section **??**). If the query cannot be evaluated, it is recorded as a postponed query. For each postponed query, *Retalis* generates a set of monitors. Monitors observe memory update events. As soon as all necessary events are in place in memory instances, the query is performed. The implementation of monitors are similar to the implementation of memory instances.

The red line in Figure 16 visualizes the CPU time of the NAO application where in each second, 1000 next queries on a memory instance of size 2500 are evaluated. In addition, for each next query, a new event is generated. The green line visualizes the CPU time of a similar case where the next queries are synchronized. This experiment is conducted in a way that no query needs to be delayed. Comparing these two cases shows that when queries are not delayed, the synchronization cost is negligible.

Figure 17 shows the CPU time of four cases. In all these cases, 1000 synchronized next queries are evaluated and 1000 events are generated in each second. The red line visualizes the case where no query is delayed. The green line visualizes the case where queries are delayed for 5 seconds. In this case, the memory instance queried by a next term has not yet received the data necessary to evaluate the query. The query is performed as soon as the memory instance is updated with relevant information. There are 1000 queries per seconds, each delayed for 5 seconds. This means there exist 5000 monitors at each time. These monitors observe 1900 events processed by *Retalis* per second. We observe that for such a large number of monitors observing such a high-frequency input stream of events, the increase in CPU time is less than 30 percent.

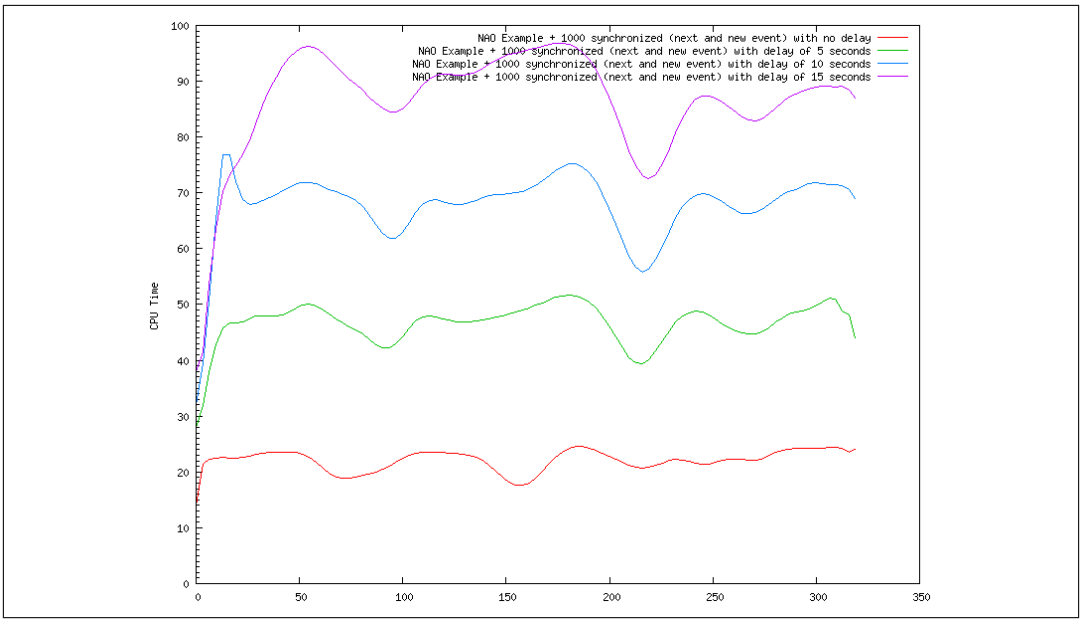

Figure 17: Synchronization with delays

6.4 On-Flow Processing

On-flow processing functionalities in *Retalis* are implemented using *ELE*. *ELE* execution model is based on decomposition of complex event patterns into intermediate binary event patterns (goals) and the compilation of goals into *goal-directed event-driven Prolog* rules. As relevant events occur, these rules are executed deriving corresponding goals progressing toward detecting complex event patterns.

Information flow processing systems such as *ELE* are designed for applications that require a real-time processing of a large volume of data flow. We refer the reader to the evaluation of the performance of *ELE* presented elsewhere [6, 3]. While the execution system of *ELE* is *Prolog*, the evaluation shows that in terms of performance, *ELE* is competitive with respect to the state-of-the-art information processing systems.

6.5 Subscription

The implementation of the subscriptions is similar to the implementation of memory instances. The only difference is that an event matching a memory instance is asserted to the knowledge base for that memory instance, and an old event is retracted if the memory instance is full, but an event matching a subscription is delivered to

the respective subscriber. Consequently, the costs of subscriptions include the unification cost, discussed in section 6.2, and the costs to publish events to subscribed *ROS* topics. The latter comprises the costs for converting events to *ROS* messages and the costs of message transportation within the *ROS* framework.

7 Conclusion

Retalis is introduced in this paper to develop information engineering components of autonomous robots. Consequently it is used for the processing and management of data to create knowledge about the robot's environment. Information engineering is an essential robotic technique to apply AI methods such as situation awareness, task-level planning and knowledge-intensive task execution. Consequently, *Retalis* addresses a major challenge to make robotic systems more responsive to real-world situations.

The *Retalis* language integrates *ELE* and *SLR*, two logic-based languages for on-demand and on-flow processing, respectively. *ELE* is used for temporal and logical reasoning, and data transformation in flow of data. *SLR* is used to implement a knowledge base maintaining history of some events. *SLR* supports state-based representation of knowledge built upon discrete sensory data, management of sensory data in active memories and synchronization of queries over asynchronous sensory data.

Retalis addresses all eight requirements discussed in the introduction. In particular, *ELE* addresses the requirements of on-flow processing, like event-driven and incremental processing, temporal pattern detection and transformation, subscription and garbage collection. *SLR* addresses the requirements of on-demand processing like memorizing, forgetting, active memory and state-based representation. In this way, *Retalis* unifies and advances the state-of-the-art research on robotic information engineering.

The contribution of this paper is threefold. The first contribution is the development of *SLR* language. *SLR* advances the state-of-the-art robotic on-demand processing systems by providing an active logic-based knowledge base. It combines the benefits of both knowledge base and active memory systems. *SLR* provides programming constructs to facilitate a high-level and efficient implementation of robotic on-demand processing functionalities. However, *SLR*, is a logic-based language based on *Prolog*. Therefore, a knowledge of *Prolog* is necessary to use *SLR*.

The second contribution is the integration of the *ELE* and *SLR* languages concerning three issues. The first issue is to process flows of sensory data on the fly by *ELE* to extract relevant knowledge for its compact storage in *SLR*. The second issue

is to query *SLR* for the knowledge built upon sensory data while processing flows of data. The third issue is to process events of changes of *SLR* memory by *ELE* to notify external components with patterns of changes that are of their interest.

The third contribution of the declarative *Retalis* language is a semantics based on a model of sensory data taking into account their occurrence times. This may be contrasted to alternative semantics based on processing times. In this way, the model captures and handles various issues related to asynchronous processing of data in robot software.

Moreover, *Retalis* is an open-source and framework-independent software library. Therefore, it can be used to empower the existing robotic frameworks with its wide range of functionalities as opposed to, for instance, robotic active memories which are usually tightly integrated with specific robotic frameworks. *Retalis* has been integrated in *ROS* and used to implement few proof-of-concept tasks for NAO robot, including data transformation, runtime subscription and high-level event detection.

A future work is to apply the machinery developed in this paper to support AI-based robotics. For example, AI research on task-level planning has developed a number of agent programming languages [17] to support the implementation of autonomous behavior based on the BDI (Belief-Desire-Intention) model of practical reasoning [18, 64, 63]. However, these languages do not support event-driven and incremental reasoning on their input data. Therefore, the sensory input processing support of these languages is not suitable for on-flow processing of data [82]. The lack of on-flow processing support reduces the reactivity and limits the application of these languages in robotics [82, 81]. Moreover, there are concerns about the performance of these languages in robotic applications. For instance, there is a performance issue caused by the repetition of queries on knowledge base. An approach to increase performance is to cache query results [1]. By caching, a query is re-evaluated only if the knowledge base has been updated with relevant facts. To implement such a caching mechanism when the agent and the knowledge base components are separated, active memory functionalities are required to inform the agent program about the changes of the knowledge base.

Other future work is to further support on-flow and on-demand processing. A work is to support the representation and reasoning about uncertain data. Very few current information engineering systems address uncertainty. This can be due to scalability issues and the training time required to learn the transition probabilities of a domain. Another work is to extend consumption, and memorizing and forgetting policies in on-flow and on-demand processing, respectively. Another work is to further support reasoning on temporally qualified knowledge. Supporting the implementation of episodic-like memories [70] is another direction of research.

References

[1] Natasha Alechina, Tristan Behrens, Mehdi Dastani, Koen Hindriks, Koen Hubner, Fred Jomi, Brian Logan, Hai H. Nguyen, and Marc van Zee. Multi-cycle query caching in agent programming. In *Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI-13)*, July 2013.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[3] Darko Anicic. Event Processing and Stream Reasoning with ETALIS. *PhD Thesis, Karlsruher Institute of Technology*, 2011.

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Epsparql: A unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[5] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems SE - 5*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin Heidelberg, 2010.

[6] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Real-time complex event recognition and reasoning - a logic programming approach. *Applied Artificial Intelligence*, 26(1-2):6–57, 2012.

[7] Krzysztof R Apt and M H van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, 29(3):841–862, July 1982.

[8] Alexander Artikis, Georgios Paliouras, François Portet, and Anastasios Skarlatidis. Logic-based representation, reasoning and machine learning for event recognition. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems - DEBS '10*, page 282, New York, New York, USA, 2010. ACM Press.

[9] Carlos Astua, Ramon Barber, Jonathan Crespo, and Alberto Jardon. Object detection techniques applied on mobile robot semantic navigation. *Sensors*, 14(4):6734–6757, 2014.

[10] Franz Baader, Ian Horrocks, and Ulrike Sattler. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier, 2008.

[11] Yaakov Bar-Shalom and Thomas E. Fortmann. *Tracking and Data Association.* Academic Press Professional, Inc, 1988.

[12] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Rec.*, 39(1):20–26, September 2010.

[13] C. Bauckhage, S. Wachsmuth, M. Hanheide, S. Wrede, G. Sagerer, G. Heidemann, and H. Ritter. The visual active memory perspective on integrated recognition systems. *Image and Vision Computing*, 26(1):5–14, January 2008.

[14] Michael Beetz, Lorenz Mosenlechner, and Moritz Tenorth. CRAM - A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017. IEEE, October 2010.

[15] Mark Birbeck. *Professional XML.* Wrox Press, 2001.

[16] Nico Blodow, Dominik Jain, Zoltan-Csaba Marton, and Michael Beetz. Perception and probabilistic anchoring for dynamic world state logging. *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pages 160–166, December 2010.

[17] Rafael H Bordini, Lars Braubach, Jorge J Gomez-sanz, Gregory O Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *INFORMATICA*, 30:33–44, 2006.

[18] Michael E Bratman. *Intention, Plans, and Practical Reason.* Cambridge University Press, March 1999.

[19] Davide Brugali and Patrizia Scandurra. Component-based Robotic Engineering Part I : Reusable building blocks. *IEEE ROBOTICS AND AUTOMATION MAGAZINE*, XX(4):1–12, 2009.

[20] Davide Brugali and Azamat Shakhimardanov. Component-based Robotic Engineering Part II : Systems and Models. *IEEE ROBOTICS AND AUTOMATION MAGAZINE*, XX(1):1–12, 2010.

[21] K. Selçuk Candan, Huan Liu, and Reshma Suvarna. Resource description framework: Metadata and its applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001.

[22] L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12(3):359–382, August 1996.

[23] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Berlin-New York: Springer-Verlag, 2003.

[24] Silvia Coradeschi and Alessandro Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, May 2003.

[25] Claudia Cruz, Luis Enrique Sucar, and Eduardo F Morales. Real-time face recognition for human-robot interaction. In *Automatic Face & Gesture Recognition, 2008. FG'08. 8th IEEE International Conference on*, pages 1–6. IEEE, 2008.

[26] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, V(i):1–70, 2012.

[27] Daniel de Leng and Fredrik Heintz. Towards on-demand semantic event processing for stream reasoning. In *Information Fusion (FUSION), 2014 17th International Conference on*, pages 1–8. IEEE, 2014.

[28] Patrick Doherty, Fredrik Heintz, and Jonas Kvarnström. Robotics, Temporal Logic and Stream Reasoning. *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-19)*, pages 42–51, 2014.

[29] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, February 2009.

[30] J. Elfring, S. van den Dries, M.J.G. van de Molengraft, and M. Steinbuch. Semantic world modeling using probabilistic multiple hypothesis anchoring. *Robotics and Autonomous Systems*, 61(2):95–105, December 2012.

[31] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[32] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013.

[33] Malik Ghallab. On Chronicles: Representation, On-line Recognition and Learning. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 597–606, 1996.

[34] Christian Halashek-Wiener, Bijan Parsia, and Evren Sirin. Description Logic Reasoning with Syntactic Updates. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[35] Nick Hawes. Building for the Future: Architectures for the Next Generation of Intelligent Robots. *Proceedings of a Symposium held in Honour of Aaron Sloman*, 2011.

[36] Nick Hawes, Aaron Sloman, and Jeremy Wyatt. Towards an integrated robot with multiple cognitive functions. *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2008), AAAI Press*, pages 1548–1553, 2008.

[37] Nick Hawes and Jeremy Wyatt. Engineering intelligent information-processing systems with CAST. *Advanced Engineering Informatics*, 24(1):27–39, 2010.

[38] Fredrik Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework.* PhD thesis, Linköping Studies in Science and Technology. Dissertations #1240. Linköping University Electronic Press. 258 Pages., 2009.

[39] Fredrik Heintz. Semantically grounded stream reasoning integrated with ROS. *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 5935–5942, 2013.

[40] Fredrik Heintz, J Kvarnström, and Patrick Doherty. Stream-Based Reasoning Support for Autonomous Systems. *European Conference on Artificial Intelligence (ECAI)*, 2010.

[41] Fredrik Heintz, Jonas Kvarnstrom, and Patrick Doherty. A stream-based hierarchical anchoring framework. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5254–5260. IEEE, October 2009.

[42] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Bridging the sense-reasoning gap: DyKnow - Stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, January 2010.

[43] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Stream-Based Hierarchical Anchoring. *KI - Künstliche Intelligenz*, 27(2):119–128, March 2013.

[44] Fredrik Heintz and D De Leng. Semantic information integration with transformations for stream reasoning. *International Conference on Information Fusion (FUSION 2013)*, 2013.

[45] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, W3C, 2004.

[46] Dominik Jain, Lorenz Mosenlechner, and Michael Beetz. Equipping robot control programs with first-order probabilistic reasoning capabilities. In *2009 IEEE International Conference on Robotics and Automation*, pages 3626–3631. IEEE, May 2009.

[47] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

[48] Pat Langley, John E. Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, June 2009.

[49] Séverin Lemaignan. Grounding the Interaction: Knowledge Management for Interactive Robots. *PhD Thesis, Laboratoire d'Analyse et d'Architecture des Systèmes (CNRS) - Technische Universität München*, 2012.

[50] Séverin Lemaignan, Raquel Ros, E. Akin Sisbot, Rachid Alami, and Michael Beetz. Grounding the Interaction: Anchoring Situated Discourse in Everyday Human-Robot Interaction. *International Journal of Social Robotics*, 4(2):181–199, November 2011.

[51] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998.

[52] Gi Hyun Lim, Il Hong Suh, and Hyowon Suh. Ontology-Based Unified Robot Knowledge for Service Robots in Indoor Environments. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 41(3):492–509, May 2011.

[53] J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc. New York, NY, USA, November 1984.

[54] I Lütkebohle. Facilitating re-use by design: A filtering, transformation, and selection architecture for robotic software systems. *ICRA'09 Workshop on Software Engineering for Robotics IV*, (section III), 2009.

[55] I Lütkebohle, R Philippsen, V Pradeep, E Marder-Eppstein, and S Wachsmuth. Generic middleware support for coordinating robot software components: The Task-State-Pattern. *Journal of Software Engineering for Robotics (JOSER)*, 2(1):20–39.

[56] Nikolaos Mavridis and Deb Roy. Grounded Situation Models for Robots: Where words and percepts meet. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4690–4697. IEEE, October 2006.

[57] Sean Bechhofer Frank van Harmelen James Hendler Ian Horrocks Deborah L. McGuinness Peter F. Patel-Schneider Mike Dean, Guus Schreiber and Lynn Andrea Stein. OWL Web Ontology Language Reference. Technical report, W3C, 2004.

[58] Federico Pecora, Marcello Cirillo, Francesca Dell Osa, Jonas Ullberg, and Alessandro Saffiotti. A constraint-based approach for proactive, context-aware human support. *Journal of Ambient Intelligence and Smart Environments*, 4:347–367, 2012.

[59] Christian Peters, Thomas Hermann, and Sven Wachsmuth. User Behavior Recognition For An Automatic Prompting System - A Structured Approach based on Task Analysis. *Proceedings of the 1st Int. Conf. on Pattern Recognition Applications and Methods (ICPRAM)*, 2:171, 2012.

[60] Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors. *Proceedings of the ICLP'07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, ALPSWS 2007, Porto, Portugal, September 13th, 2007*, volume 287 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[61] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. *Open Source Software Workshop of IEEE International Conference on Robotics and Automation (ICRA), 2009*, 2009.

[62] Surangika Ranathunga, Stephen Cranefield, and Martin Purvis. Identifying Events Taking Place in Second Life Virtual Environments. *Applied Artificial Intelligence*, 26(1-2):137–181, January 2012.

[63] Anand S Rao and Michael P Georgeff. Modeling Rational Agents within a BDI-Architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[64] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319, 1995.

[65] C Bauckhage S. Wrede, M. Hanheide, Sagerer, and G. An active memory as a model for information fusion. *International Conference on Information Fusion, Stockholm, Sweden*, 1:198–205, 2004.

[66] L. Sabri, A. Chibani, Y. Amirat, and G. P. Zarri. Narrative reasoning for cognitive ubiquitous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, 2011.

[67] Murray Shanahan. The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer, 1999.

[68] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[69] Yale Song, David Demirdjian, and Randall Davis. Continuous body and hand gesture recognition for natural human-computer interaction. *ACM Transactions on Interactive Intelligent Systems*, 2(1):1–28, March 2012.

[70] Dennis Stachowicz and Geert-Jan M Kruijff. Episodic-Like Memory for Cognitive Robots. *IEEE Transactions on Autonomous Mental Development*, 4(1):1–16, March 2012.

[71] Mori Tenorth and Michael Beetz. KNOWROB - knowledge processing for autonomous personal robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4261–4266. IEEE, October 2009.

[72] Moritz Tenorth and Michael Beetz. Knowledge Processing for Autonomous Robot Control. *Proceedings of the AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI. Stanford, CA: AAAI Press, 2012*, 2012.

[73] Moritz Tenorth, Alexander Clifford Perzylo, Reinhard Lafrenz, and Michael Beetz. The RoboEarth language: Representing and exchanging knowledge about actions, objects, and environments. *2012 IEEE International Conference on Robotics and Automation*, (3):1284–1289, May 2012.

[74] André Ückermann, Robert Haschke, and Helge Ritter. Real-Time 3D Segmentation of Cluttered Scenes for Robot Grasping. *IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012), Osaka, Japan*, 2012.

[75] Vangelis Vassiliadis, Jan Wielemaker, and Chris Mungall. Processing OWL2 ontologies using Thea: An application of logic programming. In *OWLED*, volume 529, 2009.

[76] V Verma and A Jónsson. Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. *American Institute of Aeronautics and Astronautics Space Conference*, pages 1–19, 2006.

[77] David E. Watson. Book review: Blackboard Architectures and Applications Edited by V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum (Academic Press). *ACM SIGART Bulletin*, 1(3):19–20, October 1990.

[78] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[79] R. Wood, P. Baxter, and T. Belpaeme. A review of long-term memory in natural and synthetic systems. *Adaptive Behavior*, 20(2):81–103, December 2011.

[80] S Wrede. An information-driven architecture for cognitive systems research. *Ph.D. dissertation, Faculty of Technology - Bielefeld University*, 2009.

[81] Pouyan Ziafati, Mehdi Dastani, John-Jules Meyer, and Leendert van der Torre. Agent Programming Languages Requirements for Programming Autonomous Robots. *ProMAS 2012, Springer, Heidelberg*, LNAI 7837:35–53, 2013.

[82] Pouyan Ziafati, Mehdi Dastani, John-Jules Meyer, and Leendert van der Torre. Event-Processing in Autonomous Robot Programming. *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems*, pages 95–102, 2013.

[83] Pouyan Ziafati, Yehia Elrakaiby, Mehdi Dastani, Leendert van der Torre, Marc van Zee, John-Jules Meyer, and Holger Voos. Reasoning on Robot Knowledge from Discrete and Asynchronous Observations. *AAAI Spring Symposium on Knowledge Representation and Reasoning in Robotics, Stanford, 2014*, 2014.

[84] Pouyan Ziafati, Fulvio Mastrogiovanni, and Antonio Sgorbissa. Fast Prototyping and Deployment of Context-Aware Smart Outdoor Environments. *2011 Seventh International Conference on Intelligent Environments*, pages 206–213, July 2011.