Reasoning on Robot Knowledge from Discrete and Asynchronous Observations

Pouyan Ziafati and Yehia Elrakaiby and Marc van Zee and Leendert van der Torre and Holger Voos
SnT, University of Luxembourg

Mehdi Dastani and John-Jules Meyer
Intelligent Systems Group, Utrecht University
{M.M.Dastani, J.J.C.Meyer}@uu.nl

Abstract

Robot knowledge of the world is created from discrete and asynchronous events received from its perception components. Proper representation and maintenance of robot knowledge is crucial to enable the use of robot knowledge for planning, user-interaction, etc. This paper identifies some of the main issues related to the representation, maintenance and querying of robot knowledge built upon discrete asynchronous events such as event-history management and synchronization, and introduces a language for simplifying the developers’ job at making a suitable representation of robot knowledge.

Autonomous robots with cognitive capabilities such as planning, knowledge intensive task execution and human interaction need to maintain and reason on knowledge about their environment (Beetz, Mosenlechner, and Tenorth 2010; Ziafati et al. 2013a; Tenorth and Beetz 2012; Lemaignan et al. 2011). Robot knowledge is collected by its perception components, which continuously process input sensory data and asynchronously output the results in the form of events representing various information types, such as recognized objects, faces and robot position (Heintz, Kvarnström, and Doherty 2010; Wrede 2009).

Robot knowledge, consisting of events representing observations made by its perception components, need to be properly represented and maintained to reason about it. However, the discrete and asynchronous nature of observations and their continuity make querying and reasoning on such knowledge difficult and pose many challenges on its use in robot task execution. The representation, fusion and management of various sources of knowledge and the integration of different reasoning capabilities have been the focus of many projects such as logic-based knowledge bases (Tenorth and Beetz 2012; Lemaignan et al. 2010) and active memories (Wrede 2009; Hawes, Sloman, and Wyatt 2008). In this paper, we address three requirements, that are not satisfactorily supported by existing systems. These requirements are (1) representation of continuous and discrete information, (2) dealing with asynchronousity of events and (3) management of event-histories.

Building robot knowledge upon its discrete events, time-stamped with the time of their occurrence, is not always a straightforward task since events contain different information types that should be represented and dealt with 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 is reasonable to assume that the color of an object remains the same until a new observation indicating a change of the object’s color is made. 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 the far past.

To perform its tasks, the robot’s components such as its control component query its knowledge base. Queries are answered based on the knowledge inferred from events, representing observations made by perception components, processed and received over a distributed and asynchronous network. Hence, observations may be received with some delay and out of order. 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 the delay caused by the recognition process. Therefore, correct evaluation of a 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 should be delayed until after the processing of pictures of the table by the object recognition component and the reception of the results by the knowledge base.

Robot perception components continuously send their observations to the knowledge base, leading to a growth of the 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.

In this paper, we introduce a knowledge management system, so-called SLR, for robot software that aims at dealing with the aforementioned issues. SLR supports the definition of programs to create a suitable representation of robot knowledge based on its discrete and asynchronous observations. It also supports synchronizing queries to ensure that
all data necessary to answer a query is gathered before the query is answered. Furthermore, SLR provides two memory management mechanisms for the removal of outdated and unneeded data from memory.

The remainder of the paper is organized as follows. After presenting an exemplary robot software architecture, we present the syntax and semantics of the SLR language. Then we describe the usability of SLR for reasoning on robot knowledge from its discrete events. Afterwards, we continue with describing the SLR supports for memory management and synchronization of queries. Finally, we present related work and conclude.

Robot Software Architecture

A user is interacting with robot X, requesting information about objects around it. To reply to the user’s questions, the robot’s software relies on its components shown in Figure 1: The segmentation component uses a real-time algorithm such as the one presented by Uckermann et al. (Ückermann, Haschke, and Ritter 2012) to process 3D images from the robot’s Kinect camera into 3D point cloud data segments corresponding to individual objects. Its output to SLR consists of events of the form $seg(O,L)^T$, each corresponding to the recognition of an object $O$ at time $T$ at a location $L$ relative to the camera. An identifier $O$ is assigned to each event, using an anchoring and data association algorithm such as the one presented by Elfring et al. (Elfring et al. 2012), to distinguish between events corresponding to the recognition of the same object segment. When a new object segment $O$ is recognized, the segmentation component sends its corresponding data to the objRec component. The objRec component processes the data and sends an event of the form $obj(O,Type,Color)^T$ to SLR, specifying the type Type and color Color of the recognized object. The StatePublisher component generates events of the form $tf('base','world',T^F)$ and $tf('cam','base',T^F)$. These events specify the relative position $T^F$ between the robot’s base and camera coordination frames respectively at time $T$. The control component handles the interaction with the user and queries SLR for the robot’s knowledge of the world answered by SLR based on events received from perception components. The control component also controls the orientation of the head of the robot by sending commands to the Gaze component.

Figure 1: Robot’s software components

SLR Language

SLR is a knowledge management system for robot software enabling the high-level representation, querying and maintenance of robot knowledge. In particular, SLR is aimed at simplifying the representation of the discrete pieces of information received from the robot software components, and improving efficiency and accuracy of query processing by providing synchronization and event-history management mechanisms.

The input data to SLR is a stream of events, representing robot’s observations of the world. An event is a piece of sensory information, time-stamped by the component generating it. For example, the event face(Neda, 70) could mean that Neda’s face was recognized at time 28 with a confidence of 70%. An Event can also be stamped with a time interval. For example observed(Neda)[28,49] could mean that Neda’s face was continuously perceived between times 28 and 49.

The SLR language bears close resemblance to logic programming and is both in syntax and semantics very similar to Prolog, which is the most familiar logic programming system today. Therefore we first review the main elements of Prolog upon which we then define the SLR language.

In the Prolog syntax, a term is an expression of the form $P(t_1,...,t_n)$, where $P$ is a functor symbol and $t_1,...,t_n$ are constants, variables or terms. A term is ground if it contains no variables. A Horn clause is of the form $B_1 ∧... ∧ B_n → A$, where $A$ is a term called the head of the clause, and $B_1,...,B_n$ are terms or negation of terms called the body. $A ← true$ 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 (Apt and van Emden 1982) to determine whether or not a query follows from the program. A query may result in a substitution of the free variables. We use $P$ $\vdash_{SLDNF}$ $Qθ$ to denote a query $Q$ on a program $P$, resulting in a substitution $θ$.

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 (SLR Signature).** A signature $S = \{C, R, V, Z, P^s, P^e\}$ for SLR language consists of:

- A set $C$ of constant symbols.
- A set $R ⊆ \mathbb{R}$ of real numbers.
- A set $V$ of variables.
- A set $Z ⊆ R_{t>0} \cup V$ of time points
- $P^s$, a set of $P_n^s$ 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 $n \in \mathbb{N}_{n\geq2}$, disjoint with $P^s_n$.

**Definition (Term).** A static/event term is of the form $t ::= p^s_n(t_1,...,t_{n-2})/p_e^s(\frac{t_1}{t_{n-2}}) / z_1, z_2$, where $p^s_n \in P_n^s$ and $p^s_n \in P^e_n$ 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 $p_n(t_1,...,t_{n-2})^{z_1\cdot z_2}$. Moreover, an event term whose $z_1$ and $z_2$ are identical is denoted as $p_n(t_1,...,t_{n-2})^z$. 
Definition (Event). An event is a ground event term \( p_n(t_1, \ldots, t_n)^{[z_1..z_2]} \), 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.

We introduce two types of static terms, next and prev terms who respectively refer to occurrence of an event of a certain type observed right after and right before a time point, if such event exists. In the next section we will give semantics to these notions. For now, we restrict ourselves to the syntax of SLR.

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

Definition (Previous term). Given a signature \( S \), a previous term \( \text{prev}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s) \) has an event term \( p_n(t_1, \ldots, t_n)^{[z_1..z_2]} \) and a time point \( z_s \) as its arguments.

Definition (SLR Program). Given a signature \( S \), an SLR program \( D \) consists of a finite set of Horn clauses of the form \( B_1 \land \ldots \land B_n \rightarrow A \) built from the signature \( S \), where next and prev terms can only appear in body of rules and the program exudes event facts (events).

SLR Operational Semantics

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

Definition (Event Stream). An event stream \( \epsilon \) is a (possibly infinite) set of events.

Definition (Event Stream View). An event stream view \( \epsilon(z) \) is the maximum subset of event stream \( \epsilon \) such that events in \( \epsilon(z) \) have their end time before or at the time point \( z \), i.e. \( \epsilon(z) = \{ p_n(t_1, \ldots, t_n)^{[z_1..z_2]} \in \epsilon \mid z_2 \leq z \} \). By definition, the variable \( z_s \) should be instantiated when the prev clause is evaluated and an error is generated otherwise.

Definition (Knowledge base). Given a signature \( S \), a knowledge base \( k \) is a tuple \( (D, \epsilon) \) where \( D \) is an SLR program and \( \epsilon \) is an event stream defined upon \( S \).

Definition (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 \( k \vdash \text{SLR} \langle Q, z \rangle \theta \) to denote an SLR query \( \langle Q, z \rangle \) on the knowledge base \( k \), resulting in the substitution \( \theta \).

The operational semantics of SLR for query evaluation follows the standard Prolog operational semantics (i.e. unification, resolution and backtracking) (Apt and van Emden 1982) as follows: The evaluation of a query \( \langle Q, z \rangle \) given an SLR knowledge base \( k = (D, \epsilon) \) consists in performing a depth-first search to find a variables binding that enables derivation of \( Q \) from the rules and static facts in \( D \), and event facts (i.e. events) in \( \epsilon \). The result is a set of substitutions (i.e. variable bindings) \( \theta \) such that \( D \cup \epsilon \vdash \text{SLDNF} \ Q \theta \) under the condition that event terms which are not arguments of next and prev terms can be unified with event facts only if such events belong 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. Each event is time-stamped with the time of its occurrence. In a query \( \langle Q, z \rangle \), the parameter \( z \) limits query evaluation to the set of observations made up until the time \( z \). This means that the query \( \langle Q, z \rangle \) cannot be evaluated before the time \( z \), since SLR would not have received robot’s observations necessary to evaluate \( Q \).

A query \( \langle Q, z \rangle \) can be posted to SLR long after the time \( z \) in which case the SLR knowledge base contains observations made after the time \( z \). In order to have a clear semantics of queries, the SLR operational semantics applies the following rule. When evaluating a query \( \langle Q, z \rangle \), only event facts in the knowledge base are taken into account whose end times are earlier or equal to \( z \) (i.e. event facts in \( \epsilon(z) \)). The only exception is for the case of next or prev clauses which are evaluated based on their declarative definitions regardless of the \( z \) parameter of the query as follows.

The \( \text{prev}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s) \) term unifies \( p_n(t_1, \ldots, t_n)^{[z_1..z_2]} \) with an event \( p_n(t_1', \ldots, t_n')^{[z_1'..z_2']} \) in \( \epsilon(z) \) such that \( z_s \geq z_2' \) and there is no other such event in \( \epsilon(z) \) which has the end time later than \( z_2' \). If such a unification is found, the prev clause succeeds and fails otherwise.

A prev clause \( \text{prev}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s) \) is evaluated using the following rule over the event facts in \( \epsilon \). By definition, the variable \( z_s \) should be already instantiated when the prev clause is evaluated and an error is generated otherwise.

It is also worth noting that the prev clause can be evaluated only after the time \( z_s \) when all events with end time earlier or equal to \( z_s \) have been received by and stored in the SLR knowledge base. The \( \neg \) symbols represents Prolog negation.

\[
\text{prev}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s) \leftarrow \\
\quad p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s \leq z_n \\
\quad \neg p_n(t_1', \ldots, t_n')^{[z_1'..z_2']}, z_2' \leq z_s, z_2' > z_2).
\]

The \( \text{next}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, z_s, z_e) \) term unifies \( p_n(t_1, \ldots, t_n)^{[z_1..z_2]} \) with an event \( p_n(t_1', \ldots, t_n')^{[z_1'..z_2']} \) in \( \epsilon(z_e) \) such that \( z_e \leq z_2, z_2 \leq z_e \) and there is no other such event in \( \epsilon \) which has the start time earlier than \( z_2' \). If such a unification is found, the next clause succeeds and fails otherwise.

A next clause \( \text{next}(p_n(t_1, \ldots, t_n)^{[z_1..z_2]}, \{z_s, z_e\}) \) is evaluated using the following rule over the event facts in \( \epsilon \). By definition, the variables \( z_s \) and \( z_e \) should be instantiated when the next clause is evaluated and an error is generated otherwise.

The next clause can only be evaluated after the time \( z_e \) when all 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 order of their start times, the next clause can be evaluated as
soon as the first event unifiable with \( p_n(t_1, ..., t_n) \) with start time later than \( z_n \), is received by SLR, not to postpone queries when unnecessary. This holds when only one perception component generates events of type \( p_n(t_1, ..., t_n) \) and those events are sent to SLR in order of their start time.

\[
next(p_n(t_1, ..., t_n) \mid z_1, z_2) \leftarrow p_n(t_1, ..., t_n) \mid z_1, z_2, z_e),
\]

\[
\neg p_n(t_1, ..., t_n) \mid z_1, z_2, z_e, z_a < z_1, z_2 \leq z_e.
\]

**Persistence and Maintaining State**

Robot knowledge of the world is constructed from observations made by its perception components. These observations take the form of discrete events, stamped with the time of their occurrence. This representation of robot knowledge makes the formulation of queries over the knowledge base difficult. For example, if an object was observed at some location \( L \) at time 2 and again at the same location at time 4, then a query about whether the object was at \( L \) at time 3 will fail since the knowledge base does not include an event indicating the location of the object at time 3.

SLR, by enabling the definition of programs, aims at simplifying the task of the programmer of making a suitable representation of robot knowledge. In particular, programs are meant to enable transforming the event-based representation of robot knowledge, i.e., events, into a state-based representation of knowledge, using derived facts. This section discusses some of the typical cases where a state-based representation is more suitable and how it can be specified.

**Persistent Knowledge**

Persistent knowledge refers to information such as color of an object that is assumed not to change over time.

**Example.** 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.

\[
\text{color}(O, C)^T \leftarrow \text{prev}(\text{obj}(O, C)^Z, T).
\]

**Persistence with Temporal Validity**

The temporal validity of persistence means the period of time during which it is safe to assume that information derived from an observation remains valid. For example, when an object is observed at time \( t_1 \) at a location \( L \), it may be safe to assume that the object is at \( L \) for some time period \( \delta \) after \( t_1 \). However, after the elapse of \( \delta \), it should be considered that the location of the object is unknown since it has not observed for a long time, i.e., \( \delta \).

**Example.** To pick up an object \( O \), its location should be determined and send to a planner to produce a trajectory for the manipulator to perform the action. This task can be naively presented as a sequence of actions 1) determine the object’s location \( L \) 2) compute a manipulation trajectory \( Trj \), and 3) perform the manipulation. However, due to for example environment dynamics, 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 in SLR program can be used to determine the location of an object and its validity as follows. If the last observation of the object is within last 5 seconds, the object location is set to the location at which the object was seen last time. If the last observation was made before 5 seconds ago, the second rule specifies that the location is outdated and finally, the third rule sets the location to “never-observed”, if such an object has never been observed by the robot. The symbol \( ! \) represents Prolog cut operator. For the sake of brevity, Objects’ locations determined by these rules are relative to robot’s camera. In reality we either need to calculate object’s locations in the world reference frame, or in the case we are using relative locations, we should also encode that the movement of the robot itself invalidates locations.

\[
\text{location}(O, L)^T \leftarrow \text{prev}(\text{seg}(O, L)^Z, T), T - Z \leq 5, !.
\]

\[
\text{location}(O, \text{“outdated”})^T \leftarrow \text{prev}(\text{seg}(O, L)^Z, T), T - Z > 5, !.
\]

\[
\text{location}(O, \text{“never-observed”})^T.
\]

**Continuous Knowledge**

Continuous knowledge refers to information that takes continuous values such as a relative position between a moving robot’s camera and its base co-ordination frames. For example, to precisely position an object in the world reference co-ordination frame, the camera to base relative position at time of the recognition of the object needs to be interpolated/extrapolated based on discrete events of observations of camera to base relative position over time.

**Example.** The following rule calculates the camera to base relative position \( TF \) at a time \( T \) by interpolating from the last observation (i.e. event) of the camera to base relative position \( TF1 \) at time \( T1 \) equal or earlier than \( T \) and the first observation of the camera to base relative position \( TF2 \) at time \( T2 \) equal or later than \( T \). The \( \text{interpolate} \) term is a user defined term which performs the actual interpolation. When evaluating the camera to base position at time \( T \) using this rule, the SLR execution system postpones the evaluation of the query until it receives the first \( \text{tf}(\text{‘cam’}, \text{‘base’}, TF1)^{T1} \) event whose start time (i.e. \( T2 \)) is equal or later than \( T \).

\[
\text{tf}(\text{‘cam’}, \text{‘base’}, TF)^{T} \leftarrow - \text{prev}(\text{tf}(\text{‘cam’}, \text{‘base’}, TF1)^{T1}, T),
\]

\[
\text{next}(\text{tf}(\text{‘cam’}, \text{‘base’}, TF2)^{T2}, [T, \infty]),
\]

\[
\text{interpolate}([TF, T], [TF1, T1], [TF2, T2]).
\]

The following rule calculates the base to world relative position \( TF \) at a time \( T \) by checking whether the first \( \text{tf}(\text{‘base’}, \text{‘world’}, TF2)^{T2} \) event occurring at or later than \( T \) occurs within a second after \( T \) (i.e. within \( [T, T+1] \)). In this example, the value of \( TF \) is interpolated similar to the previous example. Otherwise, the value of \( TF \) is extrapolated solely based on the last observation of the base to world relative
position $TF_1$ at time $T_1$ equal or earlier than $T$. The SLR execution system evaluates the value of $TF$ using this rule as soon as it receives the first $tf$('base', 'world', $TF_2$)$^T$ event whose start time is equal or later than $T$ or as soon as it waits enough to assure that such an event did not occur within $[T, T+1]$. The $\rightarrow$ symbol represents Prolog “If-Then-Else” choice operator.

$$
tf$('base', 'world', $TF$)$^T : = 
prev(tf$('base', 'world', $TF_1$)$^T, T),
(next(tf$('base', 'world', $TF_2$)$^T, [T, T+1]) \rightarrow
interpolate([T, T], [TF_1, T_1], [TF_2, T_2]));
extrapolate([T, T], [TF_1, T_1]).
$$

The following rule calculates the position $WorldPos$ of an object $O$ in the world reference coordination frame by querying the knowledge base (implemented by previous two rules) for camera to base and base to world relative positions at the time $T$ at which the object was recognized at the position $RelativePos$ relative to the robot’s head camera.

$$
position(seg(O, RelPos)$^T, WorldPos) : =
tf$('cam', 'base', $TF_1$)$^T,
tf$('base', 'world', $TF_2$)$^T,
pos multiply([RelPos, $TF_1, TF_2$], WorldPos).
$$

Aggregation It is often needed to query the number of some items in a state such as the objects in the environment the robot is aware of. The formulation of these queries can be simplified by transforming the event-based representation of events into a state-based representation.

Example. The query $\{goal, t_e\}$ with the goal below gives the list $List$ of all objects used for drinking tea that the robot is aware of up to the time $t_e$, along with their positions in the world, taken from their last observations. The result is a list of $object(O, WorldPos)$ facts as specified by the template given as the first argument of the $findAll$ clause.

$$
findAll(object(O, WPos), (obj(O, Type, Color)$^T_1,
usedFor(Type, tea), prev(seg(O, $RP_{pos}$)$^T_2, t_e)
position(seg(O, $RP_{pos}$)$^T_2, WPos)), List).
$$

Event History Management

An SLR knowledge base continuously receives and stores the events generated by robot’s software components processing the robot’s sensory data. All sensory events cannot be permanently stored as the amount of information grows unbounded over the lifetime of the robot. Therefore outdated data needs to be pruned from the memory to cope with the memory limited size and to increase efficiency in evaluating queries on the knowledge base.

SLR language supports two types of event history management mechanisms allowing the programmer to specify which events should be maintained and the duration of their storage. Event history management mechanisms in SLR are configured using two special purpose types of facts:

time-buffer $tBuffer$($PT$) and count-buffer $cBuffer$($P, N$). A $tBuffer$($PT$) fact specifies that events unifiable with $P$ should be maintained in the knowledge base for $T$ seconds after their end time. A $cBuffer$($P, N$) fact in the knowledge base specifies that only the last $N$ events, unifiable with $P$ by substituting their anonymous variables represented by _`_ sign should be kept in the knowledge base. If $P$ contains non-anonymous variables, for each distinct values of those variables, a separate count-buffer is created.

To eliminate any ambiguity in query evaluation due to the memory management mechanisms, SLR requires that there should not be any count or time buffers $buffer$($P_1, X$) and $buffer$($P_2, Y$) defined whose first argument $P_1$ and $P_2$ could be unified with each other. Otherwise, one buffer could for example specify that only the last event of type $a$ needs to be maintained in the memory and the other one specify that all events of type $a$ occurring during last 300 seconds should be maintained. This property of an SLR program can be checked automatically to generate an error if it is not satisfied.

**Example.** The following time buffers specify that all $face(P)$ events with end time during the last 60 seconds and all $tf(s, d, TF)$ events with end time during the last 300 seconds should be kept in the memory and be removed otherwise.

$$
tBuffer(face(P), 60s).
tBuffer(tf(s, d, TF), 300s).
$$

The following count buffers specify that only the last event of recognition of each distinct person and only the last event of recognition of each distinct object should be kept in the memory.

$$
cBuffer(observe(face(P, _)), 1).
cBuffer(observe(obj(O, _)), 1).
$$

Having the memory management mechanisms implemented as above, the query $\{goal, now\}$ with the goal below queries the SLR knowledge base for all people and objects that the robot have observed so far. The result contains only one fact corresponding to each person or object including the last location of the observation.

$$
findAll(Item, observe(Item, L)$^T, List).
$$

Note that the non-anonymous variable $P$ in $cBuffer$($inView(face(P, _)), 1$) means buffering the last recognition of each distinct person. In contrast, $cBuffer$($observe(face(_)), 1$) means only keeping the recognition of the last person and $cBuffer$($inView(face(L)), 1$) means keeping the last recognition of each distinct person in each distinct location.

**Synchronizing Queries over Asynchronous Events**

Robot’s perception components process their sensory inputs in a distributed and parallel setting. SLR receives their results as events published asynchronously and possibly over
a computer network. When SLR evaluates a query \((\text{goal}, z)\), it needs to ensure that it has already received all relevant events with end time earlier or equal to \(z\). Intuitively, this means that when SLR is to answer a query based on robot’s observations of the world up to time \(z\), it needs to first make sure that the processing of relevant sensory inputs acquired up to \(z\) has been finished by all corresponding perception components and it has received all the results. SLR receives queries with unique IDs and answers them as soon as they can be evaluated. This means in principle postponing the evaluation of one query does not delay the evaluation of the others.

**Definition** (Event Processing Time). The processing time (i.e. \(t_p(e)\)) of an event \(e\) is the time at which the event is received and added to the SLR knowledge base.

**Definition** (Event Delay Time). The delay time (\(t_d(e)\)) of an event \(e\) is the difference between its processing time and its end time (i.e. \(t_d(p(\langle z_1, z_2 \rangle)) = t_p(p(\langle z_1, z_2 \rangle)) - z_2\)).

The delay time of an event is mainly due to the time it takes for a perception component to generate it. For example, if Neda is recognized in the picture taken at time \(t_1\) and it takes \(k\) ms for the face recognition component to process that image, the event \(\text{recognized}('Neda')^t_1\) is generated at a time \(t_1 + k\). Then this event is sent to SLR, possibly over a computer network and hence processed by SLR at some later time \(t_2\). To guarantee the correct evaluation of a query, the delay times of events needs to be taken into account. Otherwise, SLR might answer a query using incomplete information not having the complete set of events relevant to the query received and stored in its knowledge base.

**Definition** (Goal Set). The goal set of a query \((\text{goal}, z)\) for an SLR program \(D\) is the largest set of event types (i.e. event functor symbols) which the SLDNF method could possibly backtrack on event terms of such types, when evaluating \(\text{goal}\) on SLR knowledge base with the SLR program \(D\). In other words, the goal set determines the set of all event types that if events of such types are part of the \(e\), then the result of \((\text{goal}, z)\) on SLR knowledge base \((D, e)\) could be different than the case of not having those events included in \(e\). The goal set can be determined by going through all rules in \(D\) using which the \(\text{goal}\) could be possibly proven and gathering all event functor symbols appearing in bodies of those rules.

Before evaluating a query \((\text{goal}, z)\), SLR first makes sure that it has received all input events with end time up to the time \(z\) whose types are included in the goal set of \(\text{goal}\). In other words, the query can be evaluated when the full history of all event types in the goal set of \(\text{goal}\) is available up to the time \(z\) as defined below 2.

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

A query \((\text{goal}, z)\) can be evaluated only after 1- the full history of the goal set of \(\text{goal}\) up to the time \(z\) is available and 2- all \textit{next} and \textit{prev} clauses which are backtracked on when evaluating the query can be evaluated based on their declarative definitions. A \textit{prev} clause \(\text{prev}(p_n(t_1, ..., t_n)[z_1, z_2])\) can be evaluated only after the full history of events of type \(p_n(t_1, ..., t_n)[z_1, z_2]\) up to the time \(z_a\) is available. A \textit{next} clause \(\text{next}(p_n(t_1, ..., t_n)[z_1, z_2], [z_s, z_e])\) can be evaluated as soon as the first event of type \(p_n(t_1, ..., t_n)[z_1, z_2]\) with start time later than \(z_s\) is received by SLR or when the full history of such events is fully available up to time \(z_e\).

To determine when the history of events of a type \(p_n\) up to a time \(z\) is fully available, SLR can be programmed in two complementary ways. In the first way, the programmer specifies 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 the time \(z\) is fully available.

The maximum delay times of events depends on the runtime of the perception components generating such events and need to be approximated by the system developer or derived based on monitoring the system at runtime. When less 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 perception 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, usually the one generating events of the type \(p_n\). When SLR receives an event \(\text{fullyUpdated}(P_n)^z\), it considers that the history of events of the type \(p_n\) up to the time \(z\) is fully available.

**Example.** When the position of an object \(O\) in the world coordination frame at a time \(T\) is queried, the query can be answered as soon as both camera to base and base to world relative positions at the time \(T\) can be evaluated. The former can be evaluated (i.e interpolated) as soon as SLR receives the first \(tf('cam', 'base', P)\) event with a start time equal or later than \(T\). The latter can be evaluated as soon as SLR receives the first \(tf('base', 'world', P)\) event with the start time equal or later than \(T\), or when it can ensures that there is no \(tf('base', 'world', P)\) event occurred within \([T, T + 1]\).

If we assume \(t_{d_{\max}}(tf('base', 'world', P))\) is set to 0.05 seconds, SLR has to wait 1.05 seconds after \(T\) to ensure this.

The \(t_{d_{\max}}(tf('base', 'world', P))\) can be set by the system developer but it can also be set by monitoring the system runtime performance. Whenever a \(tf('base', 'world', P)\) event is processed, SLR can check its delay, the difference between its end time and its time of process, and sets the
t_{\text{max}}(\text{tf}(\text{\textquoteleft base\textquoteleft}, \text{\textquoteleft world\textquoteleft}, F)) \) to the maximum delay of such events encountered so far.

**Example.** The robot is asked to look at the *table1* and tell about the cups it sees. To answer the question, the control component controls the head of the robot to take 3D pictures of the *table1* from its left to its right side starting by the time \( t_1 \) and finishing by the time \( t_2 \). Finally, it queries SLR for the cups observed on the table as \( \text{(goal, } t_2) \) where the goal is

\[
\text{findall}(O, \text{(obj}(O, \text{\textquoteleft cup\textquoteleft}, C)^T, \text{next} \text{(seg}(O, P)^T, [t_1, t_2], L)).
\]

The query lists a set of all object segments \( O \) whose type is “cup” and are observed at least once during \([t_1, t_2]\) in \( L \). One could also calculate the objects’ positions and check whether they are on *table1*, but such details have been omitted for the sake of brevity.

To answer this query, SLR should wait until the segmentation component processes all images acquired up to the time \( t_2 \) and to receive the results. Moreover, if a new object segment is recognized, the *objRec* component processes it for its type. Therefore SLR should also wait for the *objRec* component to process all new object segments recognized in acquired pictures up to the \( t_2 \) and to receive the results. Whenever the segmentation component processes an image acquired at a time \( t \), it outputs the recognized object segments and at the end, it sends an event \( \text{finished}(\text{segmentation}) \) to SLR. Whenever SLR receives a \( \text{finished}(\text{segmentation}) \) event, it knows that it has already received all object segments up to the time \( t \) (the history of \( \text{seg}(O,P)^T \) events up to the time \( t \) is fully available). The *objectRec* component also informs the SLR, when it finishes processing its input data up to each time point. SLR processes these event signals and proceeds with evaluating the query whenever the full history of both \( \text{obj}(O,T,C)^T \) and \( \text{seg}(O,P)^T \) events up to the time \( t_2 \) is available. Then the result is sent back to the control component.

**Related Work**

There are several tools for sensory data and knowledge integration in robotics, surveys of which can be found in (Wrede 2009) and (Lemaignan 2012). A category of these tools are active memories provided for instance in IIDA (Wrede 2009) and CAST (Hawes and Hanheide 2010) frameworks. Active memories are used to integrate, fuse and store robot sensory data. These systems usually do not process or reason on knowledge themselves but employ event-based mechanisms to notify other components when contents of their memories change. When notified through events, external components query the memories, process the results and often update back the memories which in turn can activate other processes. Another category of these tools are knowledge management systems such as ORO (Lemaignan et al. 2010) and KnowRob (Tenorth and Beetz 2009). These systems are used to store and reason on logical facts. The focus of these systems are on providing common ontologies for robotic data, integration and sharing of various knowledge including common-sense knowledge and integration of various reasoning functionalities such as ontological, rule-based and spatial reasoning.

**Persistence and Maintaining State**

Dealing with the persistence of knowledge over time is an issue that has been extensively studied in the area of languages for reasoning about knowledge and change, for example in knowledge formalisms such as the event calculus (Kowalski and Sergot 1989; Shanahan 1999) and the situation calculus (Levesque, Pirri, and Reiter 1998). The SLR language, on the other hand, aims at providing a practical solution for representing robots’ knowledge based on discrete observations. SLR therefore provides means to deal with aspects not considered in these *theoretical* formalisms such as dealing with the temporal validity of data and representation of continuous knowledge. Among the robotic knowledge management systems, KnowRob applies a similar approach to ours where observations are time-stamped and the knowledge base can be queried for the whole state at different time points. For example, a qualitative relation \( \text{rel}(A,B) \) between objects \( A \) and \( B \) for an arbitrary point in a time \( T \) can be examined using \( \text{holds}(\text{rel}(A,B), T) \) term. This term is evaluated by reading the location of the last perception of the objects before time the \( T \). However, similar functionalities are to be implemented by the programmer in pure Prolog. The \text{next} and \text{prev} clauses in SLR support the programming of such functionalities by providing an easy way of referring to observations ordered in time.

**Synchronization of Queries**

A unique feature of SLR comparing to other robotic sensory data and knowledge integration systems is its support for synchronizing queries over asynchronous events from distributed and parallel processes. SLR provides two synchronization mechanisms to ensure that all sensory data up to a time point have been processed by corresponding processes and the results have been made available to SLR when a query on such data is evaluated. These mechanisms are not supported by other systems and hence need to be implemented by external components who issue queries. This lack of support obviously makes the programming of external components querying data complex. The other disadvantage is that it makes a modular integration of external processes in active memories or knowledge management systems difficult. For example, consider a component validating the recognition of “typing” action, denoting a human typing on a keyboard, by checking whether a computer is also recognized in the scene (Wrede 2009). SLR query synchronization mechanisms allow such a component to query the SLR for a recognized computer whenever this component receives an event of the recognition of the “typing” action. This component can be sure that if a computer has been recognized by the time of the query, the query result contains its corresponding information no matter which component processes data to recognize computers and how much time such a process takes.

**History Management**

Pruning outdated data from memory is a necessary functionality for any system managing and integrating robotic sensory data. CoSy and KnowRob rely on external components to prune data from their memory. In ORO, knowledge is stored in different memory profiles, each
keeping data for a certain period of time. In IDA, a scripting language is provided to program various tasks operating on the memory. These tasks are activated periodically or in response to events generated when a memory operation is performed. IDA uses this mechanism to implement a garbage collection functionality similar to the time-based history management in SLR. In SLR, flexible garbage collection functionalities are blended in the syntax of the language. This functionalities allow to specify a time period to keep the history of certain events or for example to specify that only the record of the last occurrence of certain events needs to be kept in the memory.

An early version of the synchronization and memory management mechanisms introduced in this paper are previously presented in (Ziafati et al. 2013b). However, those mechanisms are to synchronize queries over, and prune outdated data from so called memory buffers which are similar to memory items in active memory systems. In this work, these mechanisms are tightly integrated into a Prolog-based logic programming language.

**Conclusion**

The discrete and asynchronous nature of observations made by robot perception components make the representation, maintenance and querying of robot knowledge a challenging task. This paper identifies three requirements for robotic knowledge management systems related to discreteness, asynchrony and management of robot observations and introduces the SLR language to support these requirements.

SLR aims at supporting the programming tasks of 1-) reasoning on persistence, continuity and temporal validity of information, 2-) managing histories of observations to prune outdated and unnecessary data from memory and 3-) dealing with query synchronization. In particular, SLR supports state-based representation of robot knowledge through the definition of SLR programs and implements two automatic synchronization mechanisms to make query and reasoning about robot knowledge more accurate. Furthermore, SLR provides two mechanisms to enable the programmer to deal with the growth of event-histories.

**Acknowledgement**

Pouyan Ziafati is supported by FNR, Luxembourg.

**References**


