Event-Processing in Autonomous Robot Programming

Pouyan Ziafati [1,3] Mehdi Dastani [3] John-Jules Meyer [3] Leendert van der Torre [1,2]

[1] SnT, University of Luxembourg
[2] CSC, University of Luxembourg
[3] Intelligent Systems Group, Utrecht University
{pouyan.ziafati, leon.vandertorre}@uni.lu
{M.M.Dastani, J.J.C.Meyer}@uu.nl

ABSTRACT

When implementing the high-level control component of an autonomous robot, one needs to process events, generated by sensory components, to extract the information relevant to the control component. This paper discusses the lack of support for event-processing when current agent programming languages (APLs) are used to implement the control component of autonomous robots. To address this issue, the use of information flow processing (IFP) systems is proposed to support the development of event-processing components (EPCs) for an autonomous robot. The necessary interaction mechanisms between a control component and EPCs are defined. These mechanisms allow run-time subscription to events of interest, asynchronous reception of events, maintaining necessary histories of events and run-time querying of the histories. Several implementation-related concerns for these interaction mechanisms are discussed.

Categories and Subject Descriptors

I.2.11 [**ARTIFICIAL INTELLIGENCE**]: Distributed Artificial Intelligence—*Languages and structures, Intelligent agents*; D.2.1 [**SOFTWARE ENGINEERING**]: Requirements/ Specifications—*Languages, Tools*

General Terms

Languages

Keywords

Robotic Agent Languages, Event-Processing, Agent Programming Languages, Autonomous Robotics

1. INTRODUCTION

Recent advances in robotics have enabled robots to perform complex tasks such as baking a cake [2]. An important capability of an autonomous robot is to perceive its environment. The software of such a robot usually consists of various sensory components which process sensory data to information at different levels of abstraction. State of the art examples of such processes are the recognition of a human's face[1], speech [2], gesture (e.g. waving) [6], behavior (e.g. brushing teeth) [12] and identifying objects' types and positions on a kitchen table [3].

A robot's high-level control component uses sensory information to make plans to achieve the robot's goals and to control and monitor the execution of its plans. To this end, the information received from sensory components often needs to be further processed to extract the relevant knowledge for the control component. Such processes, which we commonly refer to as event-processing, include operations such as filtering sensory events based on their contents and detecting complex event patterns.

Many of the popular robotic frameworks such as ROS[4] currently do not provide much support for the implementation of required high-level event-processing operations. They only provide low-level filtering mechanisms such as topic-based publish-subscribe messaging patterns and temporal synchronizations for the processing of events[5]. In these frameworks, event-processing support is left to the tools that a developer uses to implement a control component. This paper addresses the event-processing problem when BDI-based agent programming languages are used to implement the control component of an autonomous robot.

The contribution of this paper is three-fold: 1) it discusses the lack of event-processing support in current APLs and a suitable way of addressing this issue from the software architecture point of view, 2) it proposes the use of existing information flow processing (IFP) systems, in particular the ETALIS language, to support the development of event-processing components (EPCs) for an autonomous robot, and 3) it develops basic interaction mechanisms between an autonomous robot's control component and its EPCs. The paper contributes to the overall goal of sensory information processing for autonomous robot programming.

The rest of the paper is organized as follows. Section 2 presents a running example. Section 3 describes the event-processing problem for autonomous robots. Section 4 argues the lack of support for event-processing in current APLs and discusses the suitable approach to support event-processing in the implementation of an autonomous robot. Section 5 proposes the ETALIS language to support the development

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013),* Ito, Jonker, Gini, and Shehory (eds.), May 6–10, 2013, Saint Paul, Minnesota, USA.

[1] http://www.ros.org/wiki/face_recognition
[2] http://www.ros.org/wiki/pocketsphinx
[3] http://www.ros.org/wiki/tabletop_objects
[4] http://www.ros.org
[5] http://www.ros.org/wiki/message_filters

of EPCs for an autonomous robot. Section 6 presents necessary interaction mechanisms between a robot's control component and EPCs. Section 7 discusses several implementation-related concerns for the development of proposed interaction mechanisms between a robot's control component and EPCs. Section 8 discusses the related work in APLs, robotics and IFP systems. Section 9 concludes the paper and presents future work.

2. RUNNING EXAMPLE

Consider a Mobile robot with a moving head and gripper hands (e.g. PR2 [2]) assisting a user in an indoor environment. The robot's equipment includes a laser scanner, an IMU, a stereo camera on its head and a number of microphones. The environment is equipped with sensors (e.g. RFID, IR, etc) for sensing the user activities. The robot's software components include:

- $baseTF$: its outputs are events of the form $baseTF(BTF)$, each specifying a transformation matrix BTF between the world and the robot base coordination frames.
- $camTF$: its outputs are events of the form $camTF(CTF)$, each specifying a transformation matrix CTF between the base and the camera coordination frames.
- $objRec$: its Outputs are events of the form $objRec(O, Pos)$, each specifying the recognition of an object of a type O in a position Pos relative to the head camera.
- $sndRec$: an example of its outputs is the $sndRec(crash)$ event which specifies that a "crash" sound was heard.
- $usrRec$: an example of its outputs is the $usrPos(standing)$ event which specifies that the user was perceived to be in the standing position. Another example is the $usrLoc(r1)$ event which specifies that the user was perceived to be in the room r_1.

The robot's tasks include:

- Serving-drinks: Drinks and other objects are placed on a table. To find and grasp a drink of a certain type, the robot is moving around the table and taking pictures from different positions. Events from objRec, baseTF and camTF components are processed to find such a drink and to compute its absolute position in the world coordination frame.
- Emg-care: to call emergency assistance if the user falls down. The "user-falling" situation is recognized if the user position changes from "walking" or "standing" to "lying" without having a "sitting" position in between and a "crash" sound is heard at the same time.

3. EVENT-PROCESSING IN AUTONOMOUS ROBOTICS

We distinguish between the following three sensory information processing problems for a control component. The first one is to control the behavior of sensory components by controlling their operations and operational parameters. The second one is to manage and process the data outputted by sensory components to extract information relevant to the control component. The third problem is to react to the sensory information (i.e. make and execute plans) to cope with the situations of the environment. A sensory component outputs information asynchronously or as the result of a control component's query. As we do not consider the control problem of sensory components, we treat a sensory data produced in either way as an event. By the term "event", we refer to a sensory information item such as the robot's location at a time or the types and positions of objects recognized from a picture. The problem of our concern is the processes that should be performed on the flow of events received from sensory components, before they can be consumed by the control component.

It is difficult to specify a boundary on what event-processing tasks are to be dealt with when developing a high-level control component and what can be assumed as being supported by a robotic framework. This is due to the vast robotic application areas and different sensory-processing capabilities provided by different robotic frameworks. Therefore, rather than trying to present a complete set, at this stage of our research we consider the following basic event-processing operations which are usually needed in autonomous robotic and virtual agent domains [11, 15, 13].

- Filtering: to filter out events based on their contents. E.g. filter out $objRec(O, Pos)$ events of which O is not an object type the robot is looking for.
- Pattern detection: to detect occurrences of complex events. E.g. detecting the "user-falling" pattern.
- Transformation: to apply certain functions (e.g. aggregation function such as sum or max) over certain attributes of events and project the results as new events. E.g. to find the absolute position of an object based on the corresponding $objRec(O, Pos)$, $baseTF(BTF)$ and $camTF(CTF)$ events.
- Integrating static knowledge: To integrate information from a static knowledge base in the above operations. E.g. in searching for a soft drink, filtering out the hard ones based on an ontology knowledge of drinks.

4. EVENT-PROCESSING FOR APLS

In current BDI-based agent programming languages such as 2APL [5] and GOAL [9], sensory events are processed by means of rules which generate plans in response. For example event-handling rules in 2APL are of the form $\langle atom \rangle \leftarrow \langle belquery \rangle \mid \langle plan \rangle$. Such a rule generates a specific plan as the response to an event which matches its head. The $\langle belquery \rangle$ specifies in which belief state the rule can be applied. A generated plan can consist of different actions including querying and updating a robot's belief base. Although the event-processing operations presented in Section 3 can be implemented in current APLs using such event handling rules, the lack of explicit support for their implementation can make it difficult and inefficient for the following reasons.

Concurrency: While deliberation in APLs is a cyclic process consisting of sense, reason, and act operations, event-processing is an event-driven process. Therefore event-processing operations should be naturally performed in a separate thread of execution from that of the deliberation cycle. This enables the concurrent processing of events while for example the deliberation cycle is blocked with respect to the result of an external action. Also in distributed settings (e.g. a robot's software), event-processing should be performed in different places in the network. There are various reasons for this such as to utilize the distributed processing setting and to process events in the network closer to the components generating them.

Efficient implementation: Events of interest (e.g. "user-falling") should be detected as soon as the last informa-

tion (i.e. event) necessary for their detection becomes available. Also events should be kept in memory as far as they can contribute in the construction of an event of interest and removed afterwards. E.g. receiving a sequence of $usrPos(walking)$ and $usrPos(standing)$ events, only the last event is required for detecting the "user-falling" event when the $usrPos(lying)$ event is detected. Removing unused events prevents the used memory growing unbounded and increases the efficiency as those events are no longer considered in detecting an event pattern. An efficient implementation of event-processing operations and necessary memory management mechanisms require specialized algorithms and implementation care which is far more than a trivial task to be delegated to an end user of a programming language. Furthermore, construction and possibly scheduling of plans when event-processing operations are implemented using APLs event-handling rules can cause a performance decrease.

Correct implementation: Events might be received with delays which makes a correct implementation of some event patterns difficult without having a systematic support. E.g. the "user-falling" event can be detected incorrectly because of the delayed arrival of the event informing that the user was in the "sitting" position in between changing from the "standing" to the "lying" position. To address this, one should wait for the maximum delay in possible arrival of an appropriate $usrPos(sitting)$ event before detecting the "user-falling" event. The other alternative is to detect the "user-falling" event immediately, but generate a retraction event when noticing that it has been detected by mistake due to the delayed reception of an $usrPos(sitting)$ event.

Ease of programming: implementing event-processing operations in current APLs is inconvenient as a programmer needs to implement such operations at the low level of directly working with event occurrence times. For example an event pattern composed of 5 different event types needs at least the implementation of 5 event-handling rules and many comparisons on content and temporal attributes of its composed events.

For the above reasons, event-processing requires explicit and systematic support of a language to provide a high-level abstraction for the representation of event-processing operations and an efficient implementation of such operations.

4.1 Systematic event-processing Support

Our design choice to support event-processing for an autonomous robot is to support the development of separate event-processing components (i.e. EPCs) and their interactions with the control component of a robot developed using an agent programming language, rather than tightly integrating event-processing support in such a language. One reason is that, as argued above, event-processing should be performed in a different thread of execution from that of a robot's deliberation cycle.

Furthermore, clean separation between the specification of EPCs and the robot's control component supports the separation of concerns software engineering principle. Such a separation enables the development of re-usable EPCs for an autonomous robot to be used by different control components developed for different application scenarios. In addition to increasing the re-usability, such a separation is also beneficial in multi-robots settings or when there are more than one control component. In such cases the events processed by an EPC can be consumed by more than one control component.

Moreover, enabling support for the development of EPCs and their interactions with a robot's control component is aligned with our goal of providing such support for agent programming languages in general rather than for a specific language. It also enables utilizing different event-processing languages for developing EPCs as those languages evolve.

5. EVENT-PROCESSING COMPONENTS

The event-processing problem mentioned in Section 3 is not unique to autonomous robotics. A similar problem is faced in other application domains such as environmental monitoring, intrusion detection in computer networks and stock market analysis. What all these application domains have in common is the need for real-time processing of a flow of data to extract relevant information of a domain. As the result of at least a decade of research from different research communities to address such a need, many specialized systems have been developed which can be commonly referred to as "information-flow-processing" (i.e. IFP) systems [1]. These systems provide expressive and efficient languages for the implementation of wide variety of event-processing operations including the ones mentioned in Section 3, dealing with a high-volume flow of information. Therefore rather than developing our own tool, we opt to utilize the available event-processing languages to support the implementation of event-processing components for an autonomous robot.

For a comprehensive survey of IFP systems, we refer an interested reader to [1]. The IFP system of our choice is ETALIS [4] due to its unique set of characteristics as 1) being open source, 2) having a formal semantics, 3) having one of the most expressive event-processing language among others, 4) being competitive in efficiency to other popular systems such as ESPER [6], 5) enabling reasoning over static knowledge described as a logic program, 6) supporting the representation of complex events with time interval occurrence times, and 7) addressing the problem of delayed and out-of-order arrival of events.

5.1 ETALIS Language for EP (ELE)

ELE is an expressive rule-based event-processing language allowing the representation of all possible thirteen temporal relations between time interval occurrence times of two events as defined in Allen's interval algebra[10]. It can also represent non-occurrence of an event between the occurrence of two other events. 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.

- A term $t ::= c|v|f_n(t_1, ..., t_n)|p^s_n(t_1, ..., t_n)$.

[6]http://esper.codehaus.org/

- An (static/event) atom $a ::= p_n(t_1, ...t_n)$ where p_n is a (static/event) predicate symbol and $t_1, ..., t_n$ are terms.
- An atomic event is a ground event atom, referring to an instantaneous occurrence of interest.
- A complex event is a ground event atom, referring to an occurrence with duration.
- An ELE rule is a static rule r^s or an event rule r^e.
- A static rule is a Horn clause using static predicates. 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 $p^e(t_1, .., t_n)$. Due to the space limit we refer the reader to [4] for the specification of ELE event patterns. An event rule specifies a complex event to be detected based on a pattern of the occurrence of other events and the static knowledge.
- An ELE program consists of a set of ELE rules.
- An event stream $\epsilon : Ground^e \rightarrow 2^{\mathbb{Q}^+}$ is a mapping from ground event atoms to sets of non-negative rational numbers. The input flow of events to ETALIS is modelled as an event stream. This model specifies each atomic event (i.e. ground event atom) occurs in certain time instances. E.g. $\epsilon(objRec(o, p1)) = \{1, 3\}$ means considering all events received by ETALIS as its input over its lifetime, the time points at which the event $objRec(o, p1)$ occurs are 1 and 3.

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. $\epsilon, R \models a^{\langle q_1, q_2 \rangle}$) [4].

The execution model of ELE enables the effective detection of complex events at run-time following the semantics of the language. Every time an atomic event occurs, the system updates its knowledge base, encoding which atomic events have already happened and which are missing for the completion of complex events. A complex event is detected as soon as the last event required for its completion occurs. In ETALIS, common event-processing tasks such as filtering, pattern detection, transformation and aggregation are implemented using ELE event rules as in the following examples 1 and 2.

Example 1. To find a cola drink, at each time t the robot is taking a picture of the table, $objRec$ outputs recognized objects and $baseTF$ and $camTF$ generate events of the base and the camera positions respectively. Each event is time-stamped with the time to which the sensory information it contains refers. E.g. the occurrence time t in the event $objRec(o, p)^t$ refers to the time of taking the picture in which the object o was recognized. By processing these events, Rule (1) outputs the absolute position of recognized colas by generating events of the form $colaRec(APos)$. To do this, rule (1) detects event patterns in which instances of event types $objRec(O, Pos)$, $camTF(CTF)$ and $baseTF(BTF)$ occur at the same time, specified by the operator $equals$ between these events, and an object O is "cola". The absolute position of the recognized cola in the generated $colaRec(APos)$ event is calculated as in the rule's $where$ clause. Note: The matrix multiplication syntax presented in Rule (1) is for the sake of readability. In practice it is realized using an ELE static predicate.

$$colaRec(APos) \leftarrow objRec(O, Pos)\ \ equals$$
$$camTF(CTF)\ \ equals\ \ baseTF(BTF)\ \ where$$
$$(O = cola,\ APos = BTF \times CTF \times Pos). \quad (1)$$

Example 2. rule (2) [7] implements the "user-falling" pattern as follows. If the user position changes from "walking" or "standing" directly to "lying" without changing to "sitting" position first, and the "crash" sound is detected within one second before or after the user position is detected as "lying". The intuitive meanings of the ELE operators used in this rule are as follows. $NOT(e2).[e1, e3]$ is read as $e3$ occurs after $e1$, and $e2$ does not occur between their occurrence. $(e1\ OR\ e2)$ is read as $e1$ or $e2$ occurs. $(e1\ AND\ e2).1sec$ is read as both $e1$ and $e2$ occur within a time interval of 1 second.

$$falls(U) \leftarrow NOT(usrPos(sitting))[$$
$$(\ usrPos(walking)\ \ or\ \ usrPos(standing)\ )\ \ ,$$
$$(\ usrPos(lying)\ \ and\ \ sndRec(crash)\ ).1sec\ \ ] \quad (2)$$

6. INTERACTION MECHANISMS

Due to the reasons described in Section 4.1, event-processing should be performed out of a robot's deliberation cycle. To this end, interaction mechanisms between the APL program of a robot control component and EPCs should allow event-processing operations to be performed by EPCs and their results to be presented to the control component at the right times. To this end, the control component needs to be able to both subscribe to EPCs for events of interest and query EPCs for the history of events on demand.

The interest of a robot in different events changes at runtime depending on its goals and beliefs. Therefore a dynamic subscription mechanism is required to filter the events sent to its control component based on its runtime interests. E.g. filtering events for drinks of a specific type.

Some events are not of an immediate use to the robot's control component, but they might be of its interest in some future time. Therefore an on-demand querying mechanism is required to allow the control component to query EPCs for the history of events when necessary. As an EPC is dealing with possibly an unbounded flow of events, the history of events should be kept as long as necessary. E.g. the user is moving to different rooms and corresponding $usrLoc(R)$ events are generated by the $usrRec$ component. The control component does not need to process these events. However, when it wants to serve the user's drink, it does need to know his/her location. Therefore an EPC here should always keep the last occurrence of $usrLoc(R)$ event to be accessible by the control component on-demand.

The dynamic subscription and on-demand querying mechanisms are not well supported by the current IFP systems such as ETALIS. Due to the nature of application domains for which these systems have been developed, an IFP system is usually programmed to continuously process a flow of information for detecting complex events or answering a set of continuous queries for its subscribers. Although some of these systems provide an API for runtime configuration (e.g. to add a subscriber) or on-demand queries, the provided support for interaction with such systems are usually limited and without a well-defined semantics.

[7] The actual ELE syntax is slightly different.

To enable dynamic subscription and on-demand querying mechanisms for an EPC, we support the development of an interface component (IC) which wraps the output of the EPC to manage its interaction with the robot's control component. It is worth noting that the IC is not necessarily a passive filter and it can alter the processes carried on by the EPC. Due to the space limit we do not discuss the possible runtime configuration of an EPC by its IC.

We model the output of an EPC as a multiset of complex events of the form $a^{\langle q_1,q_2\rangle}$: An EPC's output stream $s : Ground^e \rightarrow \wp(T)$ is a mapping from ground event atoms to multisets of time intervals where T is the multiset of all tuples of the form $\langle t_1, t_2\rangle$, t_i is a non-negative rational number and $\wp(T)$ is the powerset of the multiset T defined such that each of its elements is a multisubset of T. The output is a multiset of events, because a complex event can be detected by more than one event rule.

For an EPC with the output stream s and given that S is the set of all possible multisets of complex events that its language can present, the interaction mechanisms that its IC enables between the EPC and a robot control component are described in the following sections.

6.1 Dynamic Subscription

The subscription mechanism allows run-time subscriptions of a control component to its events of interest by registering subscription windows. Each subscription window specifies events of a certain type and properties occurring within a specific time interval. As the result of registering a subscription window, the control component receives the part of the corresponding EPC's output which matches the subscription window specification. The matched events are sent to the control component as they are detected by the EPC. A subscription window $s\_win^{\langle t_s,t_e,q\rangle}$ has three parameters. A start time t_s is a time point (i.e. a non-negative rational number) , an end time t_e is a time point or $+\infty$, and 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 $s\_win^{\langle t_s,t_e,q\rangle}$, if it occurs within $\langle t_s, t_e\rangle$ and it matches the query pattern q (i.e. $\exists\theta(p = q_\theta)$). The expression $\exists\theta(p = q_\theta)$ means there is a substitution which can unify p and e and makes the conditions in $Cond$ true.

A subscription window $s\_win^{\langle t_s,t_e,q\rangle} : S \rightarrow S$ is a mapping from multisets of complex events to multisets of complex events such that $s\_win^{\langle t_s,t_e,q\rangle}(s) = \{p^{\langle t_1,t_2\rangle} \in s \mid \exists\theta(p = q_\theta) \wedge t_s < t_1, t_2 < t_e\}$.

The syntax for a control component u to register a subscription window is

- $id = register(u, s\_win^{\langle t_s,t_e,q\rangle})$.

Events sent to the control component due to a subscription are accompanied by the id of the corresponding subscription window which is uniquely assigned to the window when it is registered.

6.2 On-Demand Querying

The on-demand querying mechanism allows a control component to ask an IC at run-time to keep histories of certain events and query the histories on demand. A history of certain events is kept in a data structure called a buffer window. A buffer window keeps the record of part of the corresponding EPC's output which matches its specification. By querying a buffer window from an APL program, its content at a time can be accessed. A buffer window is of the three types described below. A start time t_s, an end time t_e and a query pattern q in the following are defined as for subscription windows.

A fixed buffer window (i.e. $f\_win^{\langle t_s,t_e,q\rangle}$) at each time t represents the events which matches the query pattern q, and of which occurrence times are within $\langle t_s, t_e\rangle$ and end before t: A fixed buffer window $f\_win^{\langle t_s,t_e,q\rangle} : S \times T \rightarrow S$ is a mapping from multisets of complex events and time to multisets of complex events such that $f\_win^{\langle t_s,t_e,q\rangle}(s, t) = \{p^{\langle t_1,t_2\rangle} \in s \mid \exists\theta(p = q_\theta) \wedge t_s < t_1, t_2 < min(t, t_e)\}$.

A time-based buffer window (i.e. $t\_win^{\langle t_s,q,l\rangle}$) with a time-length l, at each time t represents the events which matches the query pattern q, and of which occurrence times start after t_s and are within the last l seconds: A time-based buffer window $t\_win^{\langle t_s,q,l\rangle} : S \times T \rightarrow S$ is a mapping from multisets of complex events and time to multisets of complex events such that $t\_win^{\langle t_s,q,l\rangle}(s, t) = \{p^{\langle t_1,t_2\rangle} \in s \mid \exists\theta(p = q_\theta) \wedge max(t_s, t - l) < t_1, t_2 < t\}$.

Example 3. $t\_win^{\langle 0,<objRec(O,Pos),O=cola>,60\rangle}$ is a time-based buffer window which at each time t (i.e. global system time), keeps the history of all events which match the query pattern $< objRec(O, Pos), O = cola >$ (i.e. events of type $objRec$ of which the recognized object is "cola") and of which the occurrence times are within the last 60 seconds and are occurred after the time point 0.

A count-based buffer window (i.e. $c\_win^{\langle t_s,q,n,H/L,agg\rangle}$) at each time t represents the events which have the n Highest/Lowest values of the aggregation attribute agg among the events which matches the query pattern q, and of which occurrence times are within $\langle t_s, t\rangle$: A count-based buffer window $c\_win^{\langle t_s,q,n,H/L,agg\rangle} : S \times T \rightarrow S$ is a mapping from multisets of complex events and time to multisets of complex events such that $c\_win^{\langle t_s,q,n,H/L,agg\rangle}(s, t) = Select_{agg}^{n(High/Low)}(\{p^{\langle t_1,t_2\rangle} \in s \mid \exists\theta(p = q_\theta) \wedge t_s < t_1, t_2 < t\})$. The operator $Select_{agg}^{N(High/Low)}$ selects a multisubset of a multiset such that members of the resulting multiset have the n Highest/Lowest values of the agg attribute in the input multiset. An aggregation attribute can be the k^{th} argument or the occurrence time (occ) of events of the input multiset. Occurrence times of events are ordered based on the end time of their time intervals.

Example 4. $c\_win^{\langle 5,<usrLoc(R),true>,2,H,occ\rangle}$ is a count-based buffer window which, at each system time t, keeps the history of the last two occurrences of $usrLoc(R)$ events of which occurrence times are after the time point 5.

The syntax to create a buffer window is:

- $id = create((f/t/c)\_win^{\langle\cdot\cdot\rangle})$.

When a buffer window is created, it is assigned a unique id which can be later used to query the content of the buffer. The syntax to query the content of a buffer window is:

- $read(ID, QP)$.

Thereby QP is a query pattern applied on the result of querying a buffer window at a time to further process it (i.e. filtering, aggregation) before it is sent to the corresponding control component.

7. PRACTICAL CONCERNS

An IC receives complex events outputted from its corresponding EPC and requests (i.e. queries) from a robot's

control component(s). The process_time (i.e. t_p) of an event or a request is the time it is processed by the IC. As the result of processing a request, the IC registers a subscription window, creates a buffer window or returns back the content of a buffer window. As the result of processing an event, the IC sends it to subscribers of the matched subscription windows and buffers it in the matched buffer windows. The IC has also a garbage collection mechanism which removes expired events from buffer windows.

The delay_time (i.e. t_d) of a request r^{t_r}, denoting a request r issued by a control component at a time t_r, is the difference between its process time and its issue time (i.e. $t\_d(r^{t_r}) = t_p(r^{t_r}) - t_r$). The delay time of a request is due to its possible network traverse from the issuing control component to the IC and its stay in the IC input queue to be processed. The maximum delay time d_{max_r} of requests is assumed to exist.

The delay_time (i.e. t_d) of a complex event $p^{\langle t_1,t_2\rangle}$ is the difference between its process time and the end of its occurrence time (i.e. $t\_d(p^{\langle t_1,t_2\rangle}) = t_p(p^{\langle t_1,t_2\rangle}) - t_2$). The maximum delay time d_{max_e} of events is assumed to exist. The delay time of a complex event is mainly due to the delay times of events required for its detection. The delay time of an atomic event is the difference between its occurrence time and the time it is processed by an EPC, which is related to the processing time required by a sensory component to generate the event, a possible network traverse to reach the EPC and staying in its queue to be processed.

The delay time of requests and events causes several implementation related issues. We first describe the basics of our implementation and then discuss about such issues by considering the following usecase.

Usecase 1. Robot is searching for cola drinks from the time 100 until 900. Its camera position $p2$ in relation to its base is constant over time. Therefore to calculate the absolute position p of a cola recognized as the $objRec(cola, p1)^t$ event, the robot only needs to know its base position $p3$ at the time t of taking the picture in which the cola is recognized ($p = p3 \times p2 \times p1$). To this end, the count-based buffer window $c\_win^{\langle 0,\langle baseTF(P3),true\rangle,1,H,occ\rangle}$ in IC keeps the last occurrence of events of the type $baseTF(P3)$ at each time. The last event of the type $baseTF(P3)$ at each time corresponds to the robot's location at that time. At the time 100, the control component u subscribes to recognized colas in the time interval $< 100, 900 >$ by sending the request $register(u, s\_win^{\langle 100,900,\langle objRec(O,P1),O=cola\rangle\rangle})$ to IC. Whenever the control component receives the $objRec(cola, p1)^t$ event, it queries the buffer window for its base position (correspond to the time t) to compute the absolute position of the recognized cola.

7.1 Implementation

Our approach to implement an IC is to follow the goal directed, event-driven approach of the ETALIS implementation as it is effective in processing a large volume flow of events (i.e. hundreds to thousands of events per second on a usual workstation) [4]. The following shows our implementation of the dynamic subscription mechanism. IC has a Prolog knowledge base containing following Rules (3) and (4).

$$\begin{aligned} sub(P(X_1,..,X_n), Cond, T_s, T_e, U, Id) :- \\ assert((goal(P(X_1,..,X_n), sub(Id, U, T_s, T_e)))), \\ create\_cond(P(X_1,..,X_n), Cond, Id, U). \end{aligned} \quad (3)$$

Whenever a $register(u, sub\_win\langle t_s, t_e, q\rangle)$ request is processed by IC (q is of type $\langle p(X_1,..,X_n), Cond\rangle$), the knowledge base is queried by the goal $sub(p(X_1,..,X_n), Cond, t_s, t_e, u, id)$. Such a goal matches the head of Rule (3), resulting in the evaluation of its body which results in asserting a fact and evaluating the *create_cond* goal. The fact states that there is a subscription window which is interested in events of type p_n and keeps the information of the subscriber u, the start and end time of the window and its id. The *create_cond* predicate records the set of conditions an event of type p_n should satisfy to match the subscription window.

$$\begin{aligned} p(X_1,...,X_n,T_1,T_2) :- \\ goal(p(X_1,..,X_n), sub(Id, U, T_s, T_e)), \\ T_s <= T_1, T_2 < T_e, check\_cond(Id, U, p(X_1,..,X_n)), \\ dlv(p(X_1,..,X_n), T1, T2, U, Id). \end{aligned} \quad (4)$$

Rule (4) is applied whenever an event of type P_n is processed (i.e. Events are posted as goals to the knowledge base). It checks if there is any subscription window for that type of event of which conditions are satisfied by the event, and delivers the event to the subscriber (i.e. control component) of such a window. *dlv* is a user-defined predicate implementing an action of a broker to send information to subscribers possibly over a network. Fixed buffer windows are processed in a similar way. The difference is that in rule (4), instead of having the *dlv* predicate, the event is recorded in the knowledge base for any fixed buffer which is matched. Time and count based buffer windows are also processed in a similar way, however their implementation needs extra processes (e.g. removing expired events from a time_based buffer) to maintain their contents based on their specifications over time.

7.1.1 Delay Times of *register* and *create* Requests

When IC processes an event, it sends it to the subscribers of matched subscription windows and records it for the matched buffer windows which exist in its knowledge base and disregards it afterwards. The problem arises when an event is processed before a subscription or a buffer window, to which the event matches, is processed. E.g. consider Usecase 1 where event $objRec(cola, p1)^{101}$ (i.e. recognized at time 101) is processed at time 102 and the control component request $register(u, s\_win^{\langle 100,900,\langle objRec(O,P1),O=cola\rangle\rangle})^{100}$ (i.e. request for subscribing to $objRec(cola, P1)$ events issued at time 100) is processed at time 103. To address this issue, an event should be kept in memory until the maximum delay time of requests (i.e d_{max_r}) to which the event can possibly match passes. Rules (5) and (6) address this issue for delayed *register* requests, similar to the way ETALIS addresses event delays. Rule (5) records events of type p_n in the knowledge base. When a subscription for events of type p_n is processed, rule (6) checks if there is any event recorded by rule (5) which matches the subscription specifications (i.e. start and end times, conditions) and send it to the subscriber of the subscription, if the event is matched. An event recorded by rule (5) should be deleted when the maximum delay time of its possible subscribers is passed.

$$p(X_1, ..., X_n, T_1, T_2) : - \\ assert((goal(sub(p(Y_1, .., Y_n)), p(X_1, .., X_n, T_1, T_2)))). \quad (5)$$

$$sub(P(X_1, .., X_n), Cond, T_s, T_e, U, Id) : - \\ goal(sub(P(X_1, .., X_n)), p(Y_1, .., Y_n, T_1, T_2)), \\ T_s <= T_1, T_2 < T_e, check\_cond(Id, U, p(X_1, .., X_n)), \\ dlv(p(X_1, .., X_n, T1, T2), U, Id). \quad (6)$$

7.1.2 Delay Times of Events

If events were processed without delays, IC could delete a subscription window from its knowledge base after the system time passes the window end time t_e. However as events are processed with delays, a subscription should be still kept in the knowledge base for the maximum possible delay times of events (i.e. d_{max_e}) after its end time.

Another problem with the delay times of events is for the case of reading the content of a buffer window by a $read(ID, QP)$ request. If the content of a buffer window at a time t is requested, such a request can not be answered at the time t as an event which belongs to the content of the buffer window at time t (based on the buffer window specification) might be processed by IC with delay in some future time. E.g. consider Usecase 1 where the control component receives event $objRec(cola, p1)^{103}$ and sends a request, to read the content of the count-based buffer window for the base position at time 103. Let's assume the robot stopped at time 102 and took a picture at time 103. To calculate the absolute position of the recognized cola, the position of the base at time 103 is required which is encoded in the event generated by $baseTF$ at time 102 when the robot stopped to take a picture. That event might be processed with delay and not yet available in the buffer window at time 103, when the control component queries the buffer window. To answer a query on the content of a buffer window at time t, IC should wait for the maximum delay time of events to be passed after t to be able to correctly answer the query.

7.1.3 Delay times of *read* Requests

When a robot's control component sends a $read(ID, QP)^t$ request, (i.e. requesting the content of a buffer window issued at time t), it queries the content of the buffer window at time t. However due to possible delay time of the request, if the IC does not keep the history of buffer window contents over time, it can not correctly answer the request. E.g. consider Usecase 1 where the control component receives event $objRec(cola, p1)^{101}$ and sends a request to read the content of the count-based buffer window for the base position at time 101. Let's assume the request is processed by IC at time 105. If the IC returns back the current content of the buffer window at time 105, it might be different from the content at time 101 requested by the control component as the robot might have moved in the meanwhile. To address this issue, the history of the content of a buffer window at a time t should be kept by IC for the maximum delay time of requests to be passed after t.

8. RELATED WORK

There is not much related work on event-processing for BDI-based agent programming languages, maybe because these languages so far have not been widely used in real-world application domains. We recognize the research in [3] and [13] as being of the most related work to ours. The work in [3] considers the application of multi-agent systems to the problem of situation management in distributed large-scale systems and proposes the extension of BDI architecture with situation management components for performing event correlation and inferring situations. The work in [13] applies agents in second life virtual environments and discusses the use of the ESPER event-processing language for identifying complex events in such environments. These works do not discuss the event-processing problem in current APLs as detailed as this paper discusses. Moreover the event-processing approach is not formalized in these works and they do not discuss the interaction mechanisms this paper presents for an event-processing component.

We are not aware of the use of event-processing languages such as ETALIS for event-processing in robotics. Perhaps the most related work to ours are event-based robotic frameworks such as CAST [7] and IDA [14]. These frameworks support the subscription of components to their events of interest based on the type and/or content of events. The IDA framework also provide some other types of event filters such as the "Frequency" filter which outputs only every nth received notification. These frameworks do not support the detection of complex event patterns and do not address the problem of delayed arrival of events. However, except the limited support provided in the form of buffer windows, the permanent storage and manipulation of information items and producing corresponding events as supported in CAST and IDA memory models are not explored in our work.

Two instances of other related work are the tf[8] package in ROS and the $DyKnow$ stream-based robotic middleware [8]. tf is a specialized tool for managing relations between different coordination frames in ROS. It subscribes to and keeps the history of changes in the position of different coordination frames and it can be queried for the transformation relation between two coordination frames at a specific time. tf cannot represent complex patterns of events, does not provide a dynamic subscription mechanism and does not support a count-based history management. The history management support in tf is similar to the time-based buffer window in our work. The work in $DyKnow$ is complementary to ours as it mostly deals with the control of sensory components. In $DyKnow$ one can specify policies to control the stream of data generated by a sensory component. For example to control the frequency of the data generation, to order data in time, to generate new data only if a sensory value is changed by a certain threshold, etc. $DyKnow$ does not support event-processing, dynamic subscription and on-demand querying mechanisms presented in our work.

Time and count based windows are common concepts in IFP systems. Such windows are used to select at each system time a set of last items of a data stream to be fed to a query (e.g. aggregation function) as its input. However to the best of our knowledge, the dynamic subscription and the on-demand querying mechanisms with formal semantics, addressing delay times of events and requests, are not presented elsewhere.

9. CONCLUSION

Our work is motivated by the aim to apply APLs to au-

[8]http://www.ros.org/wiki/tf

tonomous robot programming. We consider some of the basic event-processing requirements for implementing a control component of an autonomous robot and discuss in details the problem in addressing such requirements using current APLs. It is argued that event-processing is not well supported by existing APLs and it needs explicit support from a language. Such support should provide a high-level abstraction with a well defined semantics for expressing event-processing operations and an efficient implementation of such operations.

To support event-processing, we argue for the design choice of supporting the development of separate event-processing components and their interactions with an APL program implementing a robot's control component, rather than tightly integrating such support in current APLs. The reasons for this choice from the software architecture points of view are discussed. To support the development of event-processing components, we propose the use of existing information-flow-processing systems. In particular, the ETALIS system is proposed which provides an efficient and expressive event-processing language with a formal semantics.

We develop two basic interaction mechanisms, namely dynamic subscription and on-demand querying, between a robot's control component and an event-processing component. These mechanisms allow a control component at run-time to subscribe to an EPC for its events of interests, to receive events asynchronously, to maintain the necessary history of events in an EPC and to query such a history on-demand. These mechanisms are presented with formal semantics which take into account the occurrence times of events and the issue times of subscription and query requests distinguished from their process times. A formal semantics based on event occurrence times and request issue times relieves the programmer from dealing with the problem caused by the delayed and out of order arrival and process of events and requests. The delayed process of events and requests is common in robotics and necessary to be addressed in many event-processing tasks. We discuss several implementation-related issues that should be taken into account when the dynamic subscription and on-demand querying mechanisms are implemented.

The proposed event-processing framework is not specific to the use of APLs. It supports the development of event-processing components which implement expressive and efficient event-processing operations for an autonomous robot. The provided interaction mechanisms enables software components of the robot to query, and subscribe themselves and others to the events of event-processing components. We are currently working on a prototype of our framework to be released as a software library for ROS. It provides a user-friendly interface (XML configuration file) to subscribe an event-processing component (i.e. an ETALIS instance) to ROS topics. ROS messages received on the subscribed topics are automatically converted to ETALIS events to be consumed by the event-processing component. The library provides a means for ROS components to subscribe to, and query the events of event-processing components at runtime using ROS communication mechanisms.

Our future work is to evaluate the effectiveness of our approach for event-processing in different robotic application scenarios and to study its detailed comparison with related works for its further development. Other future work is to address uncertainty in event-processing.

10. ACKNOWLEDGMENTS

Pouyan Ziafati is supported by the National Research Fund (FNR), Luxembourg.

We thank Sergio Sousa for his contribution in our prototype implementation.

11. REFERENCES

[1] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. ACM Computing Surveys Journal, 2011.

[2] M. Bollini, et al. Bakebot: Baking cookies with the pr2. In The PR2 Workshop: Results, Challenges and Lessons Learned in Advancing Robots with a Common Platform, IROS, 2011.

[3] J. Buford, et al. Extending BDI Multi-Agent Systems with Situation Management. The Ninth International Conference on Information Fusion, Italy, 2006.

[4] A. Darko. Event processing and stream reasoning with ETALIS. PhD thesis, Karlsruher Institute of Technology, 09 Nov 2011.

[5] M. Dastani. 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems, Volume 16 , Pages 214-248, 2008.

[6] U. Grobekathofer, et al. Low Latency Recognition and Reproduction of Natural Gesture Trajectories, ICPRAM, pages 154-161, 2012.

[7] N. Hawes and J. Wyatt. Engineering intelligent information-processing systems with CAST. Advanced Engineering Informatics 24(1):27-39, 2010.

[8] F. Heintz, et al. Bridging the sense-reasoning gap: DyKnow- stream-based middleware for knowledge processing. Journal of Advanced Engineering Informatics, 24(1):14-25, 2010.

[9] K. Hindriks. Programming rational agents in GOAL. Multi-Agent Programming: Languages and Tools and Applications, pages 119-157, 2009.

[10] F. A. James. Maintaining knowledge about temporal intervals. Communications of the ACM, 26:832-843, 1983.

[11] I. Lutkebohle, et al. Facilitating re-use by design: A filtering, transformation, and selection architecture for robotic software systems. Software Development and Integration in Robotics @ ICRA, 2009.

[12] C. Peters, et al. User Behavior Recognition for an Automatic Prompting System - A Structured Approach based on Task Analysis, ICPRAM, pages 162-171, 2012.

[13] S. Ranathunga, et. al. Identifying events taking place in second life virtual environments. Applied Artificial Intelligence: An International Journal, 26(1-2):137-181, 2012.

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

[15] P. Ziafati, et. al. Agent Programming Languages Requirements for Programming Cognitive Robots. Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems, ProMAS @ AAMAS, Pages 39-54, 2012.