



PhD-FSTC-2015-32
The Faculty of Sciences, Technology
and Communication

Department of Information and
Computing Sciences

# DISSERTATION

Defense held on 24/06/2015 in Utrecht

to obtain the degree of

## DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

## EN INFORMATIQUE

AND

## DOCTOR AAN DE UNIVERSITEIT UTRECHT

## IN INFORMATICA

by

POUYAN ZIAFATI
Born on 12 August 1984 in Hamedan, Iran

# INFORMATION ENGINEERING IN AUTONOMOUS ROBOT SOFTWARE

# Dissertation defense committee

Dr Holger Voos, Chairman
*Professor, Université du Luxembourg*

Dr Mehdi Dastani, Vice Chairman
*Universiteit Utrecht*

Dr Leon van der Torre, dissertation supervisor
*Professor, Université du Luxembourg*

Dr John-Jules Meyer, dissertation supervisor
*Professor, Universiteit Utrecht*

Dr Catholijn Jonker
*Professor,  Technical University of Delft*

Dr Juergen Dix
*A-Professor, TU Clausthal*

Dr Antonio Sgorbissa
*A-Professor, Universita di Genova*

*"Join me in the pure atmosphere of gratitude For life"*

Hafez

# *Acknowledgements*

A PhD thesis is not only a project you do, but also an influential part of the life you live. As such, no one can go through it alone. I would like to thank everybody who shared with me this part of life.

What is a PhD? The answer is perhaps reading and writing, some hundreds of pages! This however proved not to be so trivial. I owe much to my supervisors Leon van der Torre, John-Jules Meyer and Mehdi Dastani, and to Holger Voos who kindly joined the team for this achievement. I have learned a lot from them, not only technically, but also personally. I am very thankful for their dedication, support and encouragement, and for their patience and optimism to let me evolve. It has been an honor to work with them.

The members of the ICR group in Luxembourg and Intelligent Systems group in Utrecht have contributed immensely to my personal and professional time at Luxembourg and Utrecht. The groups have been a source of friendships as well as good advice and collaboration. Many thanks to Silvano Colombo tosatto, Diego Agustin Ambrossio, Tjitze Rienstra, Amir Saeidi, Max Knobbout, Yehia Elrakaiby, Marc van Zeel, Masoud Tabatabaei, Sina Maleki, Paolo Turrini, Emil Weydert, Wojtek Jamroga, Dragan Doder, Ana-Maria Simionovici, Joost van Oijen, Bas Testerink, Eric Kok and Gennaro Di Tosto. And special thanks to Mikolaj Podlaszewski for being such a great officemate and friend.

I gratefully acknowledge the funding source that made my PhD work possible. I was funded by the Fonds National de la Recherche Luxembourg for these four years.

Lastly, I would like to thank my family for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits. For my brothers Pedram and Parham who have always been there for me. And most of all, for my loving, encouraging, and patient wife Aida whose unconditional support during the final stages of this Ph.D. is so appreciated. Thank you.

Pouyan Ziafati
Luxembourg Ville
24 March 2015

# Contents

**Acknowledgements** iii

**Contents** iv

**List of Figures** vii

**List of Tables** ix

**1 Introduction** 1
- 1.1 Artificial Intelligence and Robotics: The Reintegration . . . 1
- 1.2 Background . . . 4
  - 1.2.1 Robotic Frameworks and Active Memories . . . 5
  - 1.2.2 Robotic Knowledge Bases . . . 10
  - 1.2.3 Agent Programming Languages . . . 13
  - 1.2.4 Requirements . . . 14
  - 1.2.5 Summary of State-of-the-Art . . . 18
- 1.3 Research Questions . . . 21
- 1.4 Methodology . . . 23
  - 1.4.1 On-Flow Processing . . . 24
    - 1.4.1.1 *ETALIS* Language for Event-Processing (*ELE*) . . . 25
  - 1.4.2 On-Demand Processing . . . 26
  - 1.4.3 Incremental Query Evaluation . . . 28
  - 1.4.4 Mix of Approaches . . . 29
  - 1.4.5 Plan Representation and Execution in Agent Programming . . . 30
- 1.5 Thesis Layout . . . 30

**2 *Retalis* Language for Robotic Information Engineering** 31
- 2.1 Running Example . . . 31
- 2.2 Architectural Overview . . . 33
- 2.3 Summary . . . 39

**3 On-Flow Information Processing** 41
- 3.1 On-Flow Processing Requirements . . . 42
- 3.2 On-Flow Processing Systems . . . 44
- 3.3 *ETALIS* Language for Event-Processing (*ELE*) . . . 45
  - 3.3.1 ELE Syntax . . . 46

3.3.2 ELE Semantics . . . 47
3.4 Runtime Subscription in *Retalis* . . . 52
3.5 Performance . . . 53
3.6 Related Work . . . 54
3.7 Summary . . . 56

**4 On-Demand Information Processing** **58**
4.1 On-Demand Processing Requirements . . . 59
4.2 *SLR* Language for Event Management and Querying . . . 60
4.2.1 *SLR* Syntax . . . 61
4.2.2 *SLR* Semantics . . . 62
4.2.3 State-Based Knowledge Representation . . . 65
Persistent Knowledge . . . 65
Persistence with Temporal Validity . . . 65
Continuous Knowledge . . . 66
4.2.4 Active Memory . . . 67
4.2.5 Synchronizing Queries over Asynchronous Events . . . 70
4.3 Related Work . . . 73
4.4 Summary . . . 77

**5 Retalis Performance** **79**
5.1 NAO application . . . 79
5.2 Evaluation . . . 81
5.2.1 Forgetting and Memorizing . . . 83
5.2.2 Querying . . . 87
5.2.3 On-Flow Processing . . . 92
5.2.4 Subscription . . . 93
5.3 Summary . . . 93

**6 Active Queries** **94**
6.1 Introduction . . . 95
6.2 Definite Logic Programs . . . 97
6.2.1 SLD Resolution . . . 97
6.3 ELE Execution Model . . . 99
6.3.1 Event-Driven Backward Chaining Rules . . . 102
6.4 Active Queries: Naive Approach . . . 106
6.4.1 Evaluation . . . 109
6.4.1.1 Belief Query Evaluation in 2APL . . . 109
6.4.1.2 Naive Approach for Data-Driven Belief Query Evaluation in 2APL . . . 111
6.4.1.3 Empirical Results . . . 112
6.5 Active Queries: Optimized Approach . . . 115
6.5.1 Incremental Update . . . 122
6.5.2 Comparison with Naive Approach . . . 123
6.6 Active Queries: Tabled Optimized Approach . . . 124
6.6.1 Tabled Logic Programming . . . 124
6.6.1.1 Incremental Evaluation of Tabled Logic Programs . . . 125

6.6.2 Tabled Optimized Approach . . . 128
6.6.3 Evaluation . . . 129
6.7 Related Work . . . 134
6.8 Summary . . . 135

**7 RobAPL Agent Programming Language 137**
7.1 Plan Execution Control Requirements . . . 137
7.1.1 Usecase Scenario . . . 138
7.1.2 Complex Plan Execution Control . . . 139
7.1.3 Plan Execution Coordination . . . 140
7.2 RobAPL: a Robotic Agent Programming Language . . . 141
7.2.1 RobAPL Architecture . . . 141
7.2.2 RobAPL Plan Overview . . . 143
7.2.3 RobAPL Plan Execution Operational Semantics . . . 145
7.3 Information Engineering Requirements . . . 147
7.3.1 Information Engineering in RobAPL . . . 149
7.4 Related Work . . . 150
7.5 Summary . . . 152

**8 Summary 153**
8.1 Contribution . . . 155
8.2 Conclusion and Future Work . . . 156

**A Retalis API and Tutorial 161**
A.1 Nodes . . . 161
A.2 Tutorial: Coordinate Transformation for NAO Robot . . . 162
A.2.1 Programming . . . 162
A.2.2 Input from *ROS* . . . 162
A.2.3 Event-Processing . . . 163
A.2.4 Memorizing . . . 164
A.2.5 Querying . . . 165
A.2.5.1 Accessing Memory Instances . . . 165
A.2.5.2 Query Synchronization . . . 166
A.2.6 Subscription . . . 167

**B *RobAPL* Plan Execution Atomic Transitions 168**

**Bibliography 179**

**Summary in English 193**

**Samenvatting in het Nederlands 195**

**Curriculum Vitae 197**

# List of Figures


2.1 Robot's software components . . . 32
2.2 *IEC* architecture . . . 34
2.3 An *IEC* in *ROS* architecture . . . 37

3.1 *ELE* event-processing operator examples, re-produced from [Anicic et al., 2012] . . . 48

5.1 NAO's software components . . . 80
5.2 NAO application . . . 82
5.3 NAO application with automatic conversion of messages and events . . . 83
5.4 Irrelevant memory instances . . . 84
5.5 tf(X,Y,V,Q) memory instances (1) . . . 85
5.6 tf(X,Y,V,Q) memory instances (2) . . . 86
5.7 tf(head,cam,V,Q) memory instances . . . 87
5.8 Memory instances of different types . . . 88
5.9 Next and prev terms (1) . . . 89
5.10 Next and prev terms (2) . . . 90
5.11 Next and prev terms (3) . . . 90
5.12 Synchronization with no delay . . . 91
5.13 Synchronization with delays . . . 92

A.1 Example of a *tf/tfMessage* message . . . 163
A.2 Example of a *tf/tfMessage* event . . . 163

B.1 Transitions of child and list nodes from the Inactive state . . . 169
B.2 Transitions of child and list nodes from the Waiting state . . . 170
B.3 Transitions of child and list nodes from the Waiting-Resources state . . . 171
B.4 Transitions of abort/pre-empt nodes from the Inactive state . . . 172
B.5 Transitions of pause/resume nodes from the Inactive state . . . 172
B.6 Transitions of abort/pre-empt nodes from the Waiting state . . . 172
B.7 Transitions of pause/resume nodes from the Waiting state . . . 173
B.8 Transitions of list nodes from the Executing state . . . 173
B.9 Transitions of abort/pre-empt nodes from the Executing state . . . 174
B.10 Transitions of pause/resume nodes from the Executing state . . . 174
B.11 Transitions of child nodes from the Executing state . . . 175
B.12 Transitions from the Failing state . . . 175
B.13 Transitions from the Finishing state . . . 176
B.14 Transitions from the Pausing state . . . 176
B.15 Transitions from the Deactiving state . . . 177

B.16 Transitions from the Paused state . . . . . 177
B.17 Transitions from the Iteration-Ended state . . . . . 178

# List of Tables

2.1 *ROS* message examples . . . . . 39

6.1 Performance of the caching mechanism . . . . . 114
6.2 Performance of the Naive active query mechanism . . . . . 115
6.3 Comparision of performance on Program 5 . . . . . 130
6.4 Comparision of performance on Program 6 . . . . . 131

*To my ever loving family and Aida*

# Chapter 1

# Introduction

This chapter introduces the problem of information engineering in autonomous robot software. Support of information engineering is motivated in the context of current efforts to integrate the fields of AI and Robotics. In particular, information engineering is recognized as a main requirement to make the robotic systems more responsive to situations of the environment and to apply Artificial Intelligence techniques and technologies in robotics. After the introduction, we continue with presenting the background knowledge and related work of this thesis. Then, information engineering requirements are discussed and a summary of the state-of-the-art in robotic information engineering is given. Afterwards, we lay down the research questions of this thesis and discuss the methodology followed to answer the questions. Finally, the layout of the thesis is presented.

## 1.1 Artificial Intelligence and Robotics: The Reintegration

Building autonomous robots is a central goal of Artificial Intelligence and Robotics. A prominent example of early attempts to build autonomous robots is the *Shakey the robot* project developed in early days of AI [Nilsson, 1984]. On the one hand, Shakey succeeded in using logical reasoning to plan and execute its physical actions. On the other hand, it high-lighted many challenges to be overcome for deploying robots in dynamic unstructured environments. Shakey had a limited perception and action execution capabilities tailored to operate in a simplified environment. It had also its planning process embedded at the heart of its control loop limiting its reactivity to changes of the environment. AI and Robotics have since diverged focusing on different aspects of such challenges.

Recent advances in robotic perception, navigation, and manipulation, as well as the improvement of robotic software engineering techniques have enabled robots to perform

complex tasks such as baking a cake [Bollini et al., 2011]. These advances have opened up the application of service robots in domestic, military, agriculture, health-care and entertainment domains. These applications demand ever increasing levels of intelligence and autonomy to achieve complex goals in dynamic environments. Consequently, there is a push toward the use of AI techniques such as knowledge representation and reasoning, planning, learning and human-robot interaction to address these demands.

AI researchers also are showing an increasing interest to embed and test their techniques in robots which interact in the real world. While robotics has been always a very interesting application domain for AI, the availability of affordable robots such *NAO*,[1] *Turtle-Bot*[2] and *AR.Drone*[3] with advanced hardware capabilities and the availability of open-source software such as *ROS* for basic operations of these robots have been the recent facilitating factors for using robots by AI researchers. This thesis work is an example of the existing opportunities for AI research groups to be involved in robotic research. This work has been performed within two AI research groups with no prior robotic experience using *NAO* robots and *ROS*.

With significant advancements in both AI and Robotics, there is now an ever increasing interest to bring the two fields together toward developing autonomous robots. There have been a large number of calls for special tracks and workshops on related topics from both major AI and robotics conferences and journals. There have been also a number of winter schools,[4] a wiki page[5] and a mailing list[6] to form and develop the "AI and Robotics" community. The technology moves forward quickly and there is a growing consensus that the next step in autonomous robotics is to empower robots with AI capabilities.

Robotic information engineering is the timely processing, management and querying of the robot's sensory data to create and use knowledge of the robot's environment. The timely extraction and dissemination of knowledge of the robot's environment plays a central role in autonomy and in making robotic systems more responsive to real-world situations. It is necessary in order to react to situations of the environment, and to make plans and execute and monitor the plan execution for achieving goals in dynamic environments. We consider the representation of knowledge in symbolic form which is the dominant knowledge representation approach in robotics [Lemaignan, 2012] and is essential for robots with AI capabilities such as situation awareness, task-level planning, knowledge-intensive task execution and human-robot interaction [Sabri et al.,

[1] https://www.aldebaran.com/en/humanoid-robot/nao-robot
[2] http://turtlebot.com/
[3] http://ardrone2.parrot.com/
[4] http://aass.oru.se/Agora/Lucia2013/description.html
[5] http://ai-robotics.wikispaces.com/
[6] https://groups.google.com/forum/#!forum/ai-robotics

2011, Beetz et al., 2010, Ziafati et al., 2013a, Tenorth and Beetz, 2012, Lemaignan et al., 2011].

Robotic information engineering is challenging due to the distributed, heterogeneous and parallel nature of robot software. Robot's software components continuously and asynchronously generates sensory information such as events of recognized faces[7] [Cruz et al., 2008], objects[8] [Astua et al., 2014], gestures [Song et al., 2012] and behaviors [Peters et al., 2012]. Events are discrete observations of the robot's continuous environment which need to be correlated and aggregated in time and further processed and reasoned about. Information processing includes applying logical, temporal, spatial and probabilistic reasoning techniques with inherent heterogeneity in data representation, functionality and communication model [Tenorth and Beetz, 2009, Heintz et al., 2009, Blodow et al., 2010, Lemaignan et al., 2011, Jain et al., 2009, Elfring et al., 2012, S. Wrede, M. Hanheide et al., 2004].

The aim of this thesis is to support knowledge-based information engineering in autonomous robot software. By using the keyword "knowledge-based", we refer to the ability of representing, integrating and reasoning about knowledge in processing and querying of information. A key concern to develop affordable, maintainable and reliable robot software is the support of re-usability in development of components and their composition [Brugali and Scandurra, 2009, Brugali and Shakhimardanov, 2010, Hawes, 2011]. Re-usablity is improved through identifying requirements of robot software development and providing models, tools and technologies to support the requirements. Re-usability advances robotic software engineering by reducing development, maintenance and benchmarking costs [Hawes, 2011, Lütkebohle, 2009, Lütkebohle et al., Heintz et al., 2010b, Bauckhage et al., 2008, Hawes and Wyatt, 2010].

We explore the requirements on timely processing, management and querying of information in AI-based robotics and develop the *Retalis* (*Etalis* for Robotics) language to support knowledge-based engineering of information in autonomous robot software. *Retalis* builds on top of recent advancements in logic programming on developing efficient and data-driven knowledge processing systems such as the complex event-processing system *Etalis* [Anicic et al., 2012, 2010, Anicic, 2011] and Tabled Logic Programs [Swift and Warren, 2010, Saha and Ramakrishnan, 2006a, 2003]. It supports a high-level and declarative implementation of information engineering functionalities and provides an execution system that efficiently implements these functionalities taking the asynchronicity of data into account. The language supports logical reasoning to reason about the domain and common-sense knowledge and can be interfaced with other components,

[7] http://wiki.ROS.org/face_recognition
[8] http://wiki.ROS.org/object_recognition

for instance, to support spatial reasoning. By a thorough comparison of *Retalis* and existing systems, we show that it unifies and advances the state-of-the-art research on robotic information engineering. We provide empirical performance results, including the implementation of a demo application for *NAO* robot, showing the efficiency and scalability of the language for processing and management of a large amount of sensory information in real-time.

Agent programming languages [Bordini et al., 2006, Vikhorev et al., 2010] are examples of AI-based tools that information engineering plays an important role for their applications in robotics. We discuss the planning and plan execution control requirements of these languages in robotics and present a proposal to accordingly advance the plan execution control capabilities of these languages. This proposal is the design specification of a plan execution control language called *RobAPL*, including its syntax, semantics and its integration in the deliberation cycle of a typical agent programming language. We discuss how the information engineering techniques developed in this thesis can be used for efficient planning and plan execution control to implement the *RobAPL* language.

## 1.2 Background

Information engineering is a main concern in various robotic research tasks. Robotic frameworks [Quigley et al., 2009, Heintz et al., 2010b, Wrede, 2009] support the flow of information between components by various communication models such as service-based and publish-subscribe mechanisms focusing on the decoupling of interacting components [Eugster et al., 2003]. Active memories [Bauckhage et al., 2008, S. Wrede, M. Hanheide et al., 2004, Hawes and Wyatt, 2010, Hawes et al., 2008] support recording of information from various sources as shared resources upon which distributed processes operate. Robotic knowledge representation and reasoning research [Tenorth and Beetz, 2009, Tenorth et al., 2012, Tenorth and Beetz, 2012, Lemaignan et al., 2011, Lemaignan, 2012] supports modeling and integration of various sources of knowledge and is applying, for instance, logical and spatial reasoning. Other research [Lütkebohle, 2009, Heintz et al., 2010b, 2013, Heintz and Leng, 2013, de Leng and Heintz, 2014, Heintz, 2013, Ranathunga et al., 2012, Pecora et al., 2012, Sabri et al., 2011, Buford et al., 2006] is concerned with the processing of information flows for example to filter and adapt a flow of information to the needs of its consumers or to correlate and aggregate information for anchoring [Coradeschi and Saffiotti, 2003], plan execution monitoring or recognizing high-level situations of the environment in real-time.

Various existing systems and information engineering requirements have been extensively studied in recent research on robotic frameworks [Wrede, 2009, Heintz, 2009] and

knowledge management systems [Lemaignan, 2012, Tenorth, 2011]. We therefore opt to present an overview of the recent advancements and research directions. To this end, we first present representatives of the state-of-the art robotic frameworks and knowledge management systems. These systems are most relevant to information engineering as they provide general and re-usable models, tools and technologies for processing, managing and querying of information in the robot's software. We do not aim to give a full description of these systems. We rather pay attention to the aspects relevant to how information is modeled, processed, managed, reasoned about and queried. In addition to the related work, this section also presents the agent programming languages for which the importance of information engineering to support their applications in robotics is discussed in Chapter 7.

After presenting the background knowledge and related work, we provide a taxonomy to categorize the existing systems according to their processing models and present a list of general information engineering requirements. Finally, we give a summary of the state of the art in robotic information engineering.

### 1.2.1 Robotic Frameworks and Active Memories

Robot's software is composed of a large number of components. These components provide different perceptual and actuation capabilities such as image processing, path planning and motion control. In order to cope with ever growing scale and scope of the robot's software, a wide variety of robotic frameworks has been developed to facilitate its development, re-use and maintainability. These frameworks facilitate robotic software development and reuse by component-based software development techniques. They provide standard interfaces for accessing heterogeneous robotic hardware and open-source repositories of robotic software packages. They also provide software development and monitoring tools such as programming environments.

Extensive studies of existing robotic frameworks and the requirements of information-driven and knowledge-processing frameworks for developing Cognitive and AI-based robots have been presented in Ph.D theses of S. Wrede [Wrede, 2009] and F. Heintz [Heintz, 2009]. *IDA* [Wrede, 2009, S. Wrede, M. Hanheide et al., 2004] and *DyKnow* [Heintz et al., 2010b, Heintz, 2009] are robotic frameworks developed in these research projects. In the following, we discuss *ROS* (Robot Operating System) [Quigley et al., 2009], that has become the current de-facto standard robotic framework and discuss *IDA* and *DyKnow* as representatives of state-of-the-art research on AI-based and cognitive robotic frameworks.

*ROS* is the most widely used robotic framework. The ROS repository has an ever increasing number of state-of-the-art software packages for interfacing various robotic hardware, and for performing different robotic tasks such as *Simultaneous Localization and Mapping* (*SLAM*) and image processing. The availability of advanced robotic software packages in *ROS* significantly eases the rapid prototyping and development of complex robotic applications. In *ROS*, software packages (i.e. nodes) can be developed in different languages such as C++, Python and Java. These nodes can be started, killed, and restarted at runtime and communicate with each other in a peer-to-peer fashion. *ROS* supports three models of communication among the components: synchronous service-based (i.e. request-reply) interaction, asynchronous publish-subscribe streaming of data, and key-value based storage/retrieval of data on/from a central server. In both service-based and publish-subscribe models of interaction, the interacting components are decoupled in the sense that they do not refer to each other directly. For example, publish-subscribe in *ROS* is topic-based. Components publish data on topics without knowing the receivers. Data published on a topic is received by the components subscribed to that topic. Despite this decoupling, actual transition of data among components is peer-to-peer which is important for scalability. There is a *ROS* server that observes the topics and connects the publishers and subscribers accordingly. *ROS* components communicate by exchanging messages. The format of messages is based on a simple standard language similar to C language data structures. *ROS* supports robotic simulators such as *Stage* [Gerkey et al., 2003], *Gazebo* [Koenig and Howard, 2004] and *MORSE* [Echeverria et al., 2011]. Moreover, *ROS* has been integrated with many other robotic frameworks such as *OpenRAVE* [Diankov and Kuffner, 2008], *Orocos* [Bruyninckx, 2001], and *Player* [Gerkey et al., 2003].

*Information Driven Architecture (IDA)* uses the *XML* data type for the format of messages exchanged among components. A message is a tree-structured hierarchy of elements and attributes that is self-descriptive and therefore is easy to understand for humans and to interpret by the components processing it. In *IDA*, *Xpath* queries [Birbeck, 2001] are used for path-based access to data contained in messages, enabling a high-degree of loose-coupling and a content-based communication model. Built on top of the *XML* representation and corresponding technologies, *IDA* provides a rich communication platform that goes beyond the flexibility of the *ROS* fixed topic-based

publish-subscribe platform, as described below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<OBJECT>
  <HYPOTHESIS>
    <GENERATOR>Object Recognizer BU(N)</GENERATOR>
    <RATING>
      <RELIABILITY value="0.6"/>
      <RELEVANCE value="0.5"/>
    </RATING>
  </HYPOTHESIS>
  <CLASS>Cup</CLASS>
  <REGION image="img_office210703_122">
    <RECTANGLE x="335" y="245" w="65" h="80"/>
  </REGION>
  <CENTER x="32" y="44"/>
</OBJECT>
```

LISTING 1.1: Example of a message in *IDA* [Wrede, 2009]

Listing 1.1 presents an example of a message in *IDA* informing about the recognition of an object of the type cup. In this message, "CENTER" is a data item containing information about the location of the recognized cup. Using *XPATH*, the item can be accessed by specifying the partial path expression "/*/CENTER", regardless of the actual location of the item in the message. Accessing information items without the need of specifying a direct reference increases the loose-coupling as follows. A component does not need to be able to interpret the whole message and it can handle different types of messages, as far as they contain the necessary information for its operation. For example, a control component used to track people and objects can handle messages of the recognition of people and objects which could be of different types. As long as messages contain the "CENTER" elements, the control component can read the location of the people and objects and adjust the base and camera of the robot to follow them.

IDA supports an advanced filter-based publish-subscribe communication model. In this model, a sequence of filters are defined for a subscription to limit the number of messages received by the subscriber. From messages that are published by other components, a subscriber is notified of all messages that satisfy the constraints described by the filters. In event-based frameworks such as *IDA* and *ROS*, a call-back function is registered for each subscription. For each event that matches a subscription, its corresponding callback function is called with the event being input data of the function. In addition to specifying constraints on the content of messages using *XPATH* expressions, filters can also transform their input messages into new ones.

Listing 1.2 presents an example of defining a subscription having three filters. The incoming messages are filtered for highly reliable and frequent detection of faces. When messages match this subscription, they are dispatched to the registered callback handler, which in this case appends them to a queue.

```
1 SynchronizedQueue<FaceEvent> faces = new FaceQueue();
2 Subscription s = new Subscription();
3 s.append(new TypeFilter(FaceEvent.class));
4 s.append(new XPathFilter(new XPath("//HYPOTHESIS/RATING/RELIABILITY[
      @value>=0.95]")));
5 s.append(new FrequencyFilter(10,1,TimeUnit.SECONDS));
6 // add subscription to router object
7 r.subscribe(s,new QueueAdapter<FaceEvent>(faces));
```

Listing 1.2: Example of a filter-based subscription in *IDA* [Wrede, 2009]

On top of its filter-based publish-subscribe platform, *IDA* introduces an active memory system for information fusion and to decouple the communicating components in time. An active memory stores *IDA* messages as memory elements. Memory elements can be added, deleted, updated and queried. *XPATH* queries can be used to query or delete memory elements based on their contents. The memory is called active due to intrinsic and extrinsic processes that operate on it. These processes are automatically invoked due to changes of the content of memory. After each addition, deletion or update of a memory element, an event is generated containing information about the operation. Internal and external processes can subscribe to these events using the *IDA* subscription mechanism. For instance, it is possible to specify and register a subscription that states "a notification shall be issued if a memory element of type *FaceEvent* with a recognition probability of larger than 95% has been inserted or updated at least 10 times within a second."

An example of an intrinsic process could be a forgetting process that observes the addition and update of memory elements and removes any memory elements whose reliability is less than a threshold or has not been updated for a certain amount of time. Internal processes can be developed using a scripting language and are invoked upon the occurrence of a relevant memory event. External processes are typical components that register to memory events. An example could be a component used to perform consistency validation. For instance, if a user is perceived to be performing the *typing* action, but no computer keyboard is detected, the perception could be doubted. In *IDA*, a component is developed that observes the memory and adjust the reliability of perceptions by taking into account the context accumulated in the memory.

*DyKnow* introduces a formal and declarative language called *KPL* to specify the existing components and their connections. From such a specification, the actual system is instantiated. Streams are flows of data items, each item containing information about objects, their attributes and temporal context. Streams are generated by processes and are adapted to certain policies by stream generators. A policy is a declarative specification of some requirements. For instance, a policy may specify that updates should be send in a certain frequency, or for every change of more than a threshold. In the former case, the policy can specify how to synchronize, or extrapolate the input data to generate a stream that produces new data at a given frequency.

In parallel to our work, *DyKnow* has been extended with multiple tools to automate the generation of required streams using other streams and available processes [Heintz and Leng, 2013, Heintz, 2013]. The work is motivated to evaluate logical formulas over streams. To this end, each symbol in a given formula should be mapped to a relevant feature of a stream. The approach taken is as follows. An ontology of the domain is provided. In addition, streams and the processes that can generate or operate on the streams are annotated with their semantic descriptions. Given a formula, a semantic matching operation is performed to find whether the required streams are available or how they can be generated using the available processes and streams. Required processes are instantiated and applied to relevant streams and streams are fused and synchronized to generate the required streams.

*DyKnow* has been recently extended with a tool for processing of streams for event recognition [de Leng and Heintz, 2014]. To recognize events, *DyKnow* integrates the *C-SPARQL* [Barbieri et al., 2010] language. Events are annotated with their definitions in the *C-SPARQL* language including the streams required to detect them. When detecting an event is required in a context, the system generates the required input streams of data using the semantic matching functionality, described above. The results are then converted to *RDF* triples over which the *C-SPARQL* query that defines the event can be

evaluated. An event here may refer to a class of events, of which instances are detected.

```
1  REGISTER STREAM HighSpeedEvent COMPUTE EVERY 1s AS
2  CONSTRUCT {?uav dyknow: altitude ?avgSpeed}
3  FROM <http://www.ida.liu.se/dyknow/ontology.rdf>
4  FROM STREAM <http://www.ida.liu.se/dyknow/s59.trdf>
5     [RANGE 5s STEP 1s ]
6  WHERE {
7     ?uav    a                dyknow:UAV  .
8     ?uav    dyknow:speed     ?spd  .
9           }
10 AGGREGATE {(?avgSpeed, AVG, {?spd} )
11       FILTER (?avgSpeed > 100)}
```

Listing 1.3: Example of a *C-SPARQL* query in *DyKnow* [de Leng and Heintz, 2014]

Listing 1.3 presents an example of *C-SPARQL* query that averages the speed of objects of type *UAV* in the last five seconds. The average is computed every one second. A new stream is generated that contains the name and the average speed of the objects whose average speed is greater hundred. To increase re-usability, a query can be parametrized and regarded as a template. At runtime, the query parameters, including the streams over which the query should be evaluated, can be instantiated based on the specific event to be detected. The required input streams can then be generated by a semantic matching technique and fed to the *C-SPARQL* engine to evaluate the query.

### 1.2.2 Robotic Knowledge Bases

Autonomous robots require a large amount of domain and common-sense knowledge in order to flexibly operate in unstructured environments and interact with humans. Such knowledge cannot be hard coded in the robot's control loop. Acquiring, managing, sharing and reusing knowledge require formal ontologies to provide a shared vocabulary and reasoning capabilities to reason on the pieces of knowledge to derive new facts. There has been significant work in the last few years on developing knowledge representation and reasoning systems for robotics and such systems are actively being developed, for instance, in European projects such as RoboHow.[9] and RoboEarth[10]

Extensive studies of existing robotic knowledge management systems and the requirements for knowledge representation and reasoning have been presented in Ph.D theses

[9] http://robohow.eu/project
[10] http://roboearth.org/

of S. Lemaignan [Lemaignan, 2012] and M. Tenorth [Tenorth, 2011]. Logic-based approaches for knowledge management in robotics are dominant. Examples of such systems are *ORO* [Lemaignan et al., 2011, Lemaignan, 2012], *KnowRob* [Tenorth and Beetz, 2012, 2009] and *OUR-K* [Lim et al., 2011]. These systems provide extensive ontologies to model domain and common-sense knowledge for service robots in order to acquire and integrate various sources of knowledge, share it among robots and reason about it for flexible action execution and human-robot interaction. These systems have been interfaced with various components to, for instance, perform spatial and probabilistic reasoning and acquire knowledge from Internet and observing human behavior. The following briefly presents *KnowRob* and *ORO* as representatives of the state-of-the-art robotic knowledge management systems.

*KnowRob* provides an extensive ontology of concepts for robotics. The choice of language to represent knowledge is the *Web Ontology Language* (*OWL*)[11] [Mike Dean and Stein, 2004] which is based on *Description* logics (*DL*) [Baader et al., 2008]. *OWL* has been developed by the Semantic Web community and is widely used for knowledge representation in various domains as it provides a good level of expressiveness and supports efficient inferencing. *OWL* can represent classes, properties and relations in hierarchies. Instances of classes, their properties and their relations can be represented too. New classes can be defined as, for instance, intersection or union of other classes, or by specifying restrictions on their properties and relations. Properties can be defined as symmetric, transitive and so forth. The *KnowRob* ontology includes a large taxonomy to describe events, temporal information, objects, actions, tasks, processes, robots' capabilities, etc. For example, an event is a temporal thing which, for instance, can be used to describe the perception of an object. An event may have properties such as location, start time and end time.

*KnowRob* uses *Prolog*, described in Section 4, as its inference engine by giving the following reasons [Tenorth and Beetz, 2012]. Classical description logics reasoners always keep a classified version of the knowledge base and everything needs to be re-classified whenever the knowledge base is updated. Robots continuously sense to update their knowledge of the environment. Consequently, the re-classification of knowledge after every update can cause performance issues. Another reason is to enjoy the benefits of the closed-world assumption in *Prolog*. The assumption is that, in contrast to the open-world assumption in *DL* reasoners, everything which is not known to be true is false. While this conflicts with the usual *DL* semantics, it leads to a more compact knowledge representation as absence of things does not need to be described and the lack of knowledge about things can be used, for instance, to trigger action execution to acquire the relevant knowledge.

[11] http://www.w3.org/TR/2004/REC-owl-ref-20040210/

The third reason that *KnowRob* uses *Prolog* is to attach procedures to predicates which are called computable predicates. Computable predicates are used to extend the reasoning capabilities beyond description logics, to integrate external inference capabilities such as probabilistic reasoning or to acquire and integrate additional knowledge from external sources such as a perception component while executing a reasoning task. A simple example of a computable predicate is the *after* relation between two points in time. In *OWL*, *after* can be defined as a property relating two time points.

Listing 1.4 shows the implementation of the *after* predicate in *Knowrob*. The predicate checks whether the arguments have correct types, transforms the time points into numerical values using the *term_to_atom* predicate and finally checks whether the second argument is after the first one. When a system is queried whether a given observation is after the other one, corresponding time points are extracted and the relation is checked using the *after* predicate.

```
1 comp_after ( ? Pre , ? After ) : -
2    owl_has ( ? Pre , type , 'TimePoint' ) ,
3    owl_has ( ? After , type , 'TimePoint' ) ,
4    term_to_atom ( ?P , ? Pre ) ,
5    term_to_atom ( ?A, ? After ) ,
6    ?P < ?A.
```

Listing 1.4: Example of a computable predicate in *KnowRob* [Tenorth, 2011]

The presented implementation is only correct when both arguments are bound. As also noted by its author [Tenorth and Beetz, 2012], the correct implementation should be able to handle the different combinations of bound/unbound variables to be declaratively correct. For example, a query might ask for all observations which have occurred after a given observation. In general, the predicate should also implement the cases where the first argument is unbound, the second argument is unbound and both arguments are unbound. As mentioned above, computable predicates can go beyond simple numerical analysis as presented in this example. For instance, using the *C++* interface of *SWI-Prolog*, the *Prolog* system used in *KnowRob*, a computable predicate can implement an arbitrary *C++* function or acquire information from an external source. However, they should be used with care as their implementations in general are not declarative.

*ORO* uses *OWL* to represent knowledge and *Pellet*[12] [Sirin et al., 2007], a DL reasoner, for inference tasks. The *ORO* and *KnowRob* ontologies are similar, but *KnowRob* is more intensive in describing, for instance, objects, tasks and processes. In particular, time is not represented in *ORO* and therefore no temporal reasoning is possible. For

[12] http://clarkparsia.com/pellet/

instance, *ORO* relies on an external module to update the location of objects and history of the location of objects in the past is not available. In general, due to open-world assumption in description logic, the representation and reasoning on changes such as events and actions are not supported and should be handled externally [Ziafati et al., 2011, Lemaignan, 2012].

The focus of *ORO* is on human-robot interaction. In this regard, a key feature of *ORO* is storing independent cognitive models for every agent it interacts with. Each agent is assigned a separate *OWL* model to store its beliefs. Having a separate model for each agent allows to store and reason on different, possibly inconsistent, models of the world. For example, the perspective of agents can be taken into account. An object might be visible to one agent and not to another. Taking the perspective into account can be used, for instance, to dissolve ambiguities in dialogue. A module has been integrated in *ORO* that computes some symbolic properties such as *isVisible* according to perspectives of agents.

Another feature of *ORO* is allowing to attach a memory profile to each statement to govern their lifetimes. There are three types of memory profiles predefined: short term, episodic and long term. These memory profiles correspond to lifetimes of 10 seconds, five minutes and no time limit, respectively. When the lifetime of a statement is ended, it is automatically removed from the knowledge base. This mechanism implements a simple form of forgetting, the advanced version of which is implemented by the *IDA* active memory, as described above.

With regard to active memory functionalities, another distinguishing feature of *ORO* comparing to other logic-based knowledge bases is supporting an event notification mechanism. In addition to typical querying of the knowledge base, *ORO* allows components to subscribe to events. For example, a component can subscribe to events of the type *"?agent isVisible true, ?agent type Human"*. As soon as the perception layer detects a human in front of the robot and updates the knowledge base with the corresponding facts, an event is triggered and dispatched to the subscriber. Events do not need to match the explicit facts asserted to the knowledge base and can be generated if their triggering conditions are inferred to be true using *ORO* inference capabilities.

### 1.2.3 Agent Programming Languages

In order to achieve complex goals in dynamic environments, a robot needs to reason on its objectives and the state of its environment to select appropriate course of actions. Various agent programming languages [Bordini et al., 2006, Vikhorev et al., 2010] such as 2APL [Dastani, 2008], GOAL [Hindriks, 2009] and Jason [Bordini and Hübner,

2005, Píbil et al., 2012] have been developed to facilitate the implementation of such deliberative behaviour based on the BDI (Belief-Desire-Intention) architecture [Rao and Georgeff, 1995, 1991] inspired by the BDI model of human practical reasoning [Bratman, 1999].

An agent operation in the BDI architecture is the cyclic execution of a deliberation process in which the agent processes its input data, updates its goals and beliefs, applies a set of plan generating rules to plan upon its goals and beliefs and executes some of its planned actions. To achieve its goals, a BDI agent usually does not plan from scratch, but selects from a set of plan templates and instantiates them based on its context (i.e. goals and beliefs). Such *reactive planning* capability makes BDI-based agent programming languages particularly useful for programming agents such as robots operating in dynamic environments.

While agent programming languages provide a suitable level of abstraction and programming support for implementing deliberative behaviour, they reveal various shortcomings when applied in robotics [Ziafati et al., 2013a]. Consequently, the application domains of such languages so far have been mainly limited to cognitive software agents in simulated toy examples. The recent availability of affordable autonomous robots such as NAO humanoid robot and open-source robotic frameworks such as *ROS* facilitates research on applying APLs in autonomous robotics. Robotics provides an important and challenging domain to research on design and development of APLs. Moreover, APLs might facilitate the development of autonomous robots beyond the support provided by current robot programming languages.

Two main requirements to facilitate the application of agent programming languages in robotics are to better support the information engineering of the robot's sensory data and the control and coordination of the execution of its plans. A detailed discussion of these requirements is left for Chapter 7. However, after presenting the robotic information engineering requirements in the next section, we briefly discuss the information engineering support of the current agent programming languages and its implications on the usability of these languages in robotics.

### 1.2.4 Requirements

There are four general dimensions to explore functional requirements of information-engineering:

- *Data*: concerns the choice of data structure and ontologies to model and manipulate heterogeneous data. The choice of data representation directly affects the

performance, querying and reasoning capabilities of a system. For instance, while simple data structures such as *ROS* messages[13] [Quigley et al., 2009] are suitable for efficient data serialization and transportation, supporting ontological and logical reasoning requires a logic-based representation.

- *Memory*: concerns maintaining the history of sensory information in memory for timely access to it. Various memory management mechanisms are required to deal with continuous reception of information from perception components, including selective recording of information, pruning outdated information and synchronizing access to information. Some records of data should be stored permanently to be queried in the future. Other records of data should be processed in real-time to extract the required information and notify the relevant component, the consumers of the information.

- *Process* concerns the processing, reasoning and querying functionalities such as logical, temporal and spatial reasoning tasks performed on data to extract relevant information for its consumers. In particular, language support is needed to deal with the discrete nature of the robot's observations. Information from various sources should be correlated and aggregated in time to reason on the state of the environment or to detect high-level events.

- *Access* concerns models of communication among providers of information, *Information Engineering Components* (*IECs*) maintaining and processing information, and consumers. Providers often continuously process sensory information and asynchronously send their results to *IECs*. The synchronous access is when a consumer queries the information maintained by an *IEC* and waits to receive the results. To answer the query, the *IEC* can also synchronously access information or processing services from providers. The asynchronous access is when the *IEC* sends information to consumers without having them halted their usual operations to receive such information. Consumers typically subscribe to *IECs* for some information that interest them and receive notifications when such information is available.

According to the different types of *access* that consumers access the information processed and maintained by *IECs*, we distinguish between three general models of information processing: on-flow processing, on-demand processing and incremental query processing. Each of these processing models impose distinct requirements on *data*, *process* and *memory* dimensions, as follows.

[13] http://wiki.ROS.org/Messages

On-flow processing is the processing of sensory data on the fly in order to extract information about the environment. An example is the monitoring of smoke and temperature sensor readings in order to detect fire. A fire alarm should be generated if there is smoke and the temperature is high, observed by sensors in close proximity within a given time interval. A notification about fire detection is sent, for instance, to a plan execution component to react on it. We refer to receivers of the notifications as consumers. This example highlights the following on-flow processing requirements.

1. Event-driven and incremental processing: on-flow processing requires a real-time event-driven processing model. Relevant information should be derived as soon as it can be inferred from the sensory data received so far. Therefore, sensory data should be processed and reasoned about as soon as they are received by the system. Moreover, real-time processing of sensory data requires incremental processing techniques.

2. Temporal pattern detection and transformation: on-flow processing requires representing and detecting temporal patterns in flow of data and transforming data into suitable representations. The detection and transformation of data patterns are required to integrate sensory data in time and to detect high-level events occurring in the robot's environment.

3. Subscription: information derived from on-flow processing of data should be disseminated selectively. This is needed, for instance, not to overload a plan execution component with irrelevant events.

4. Garbage collection: records of data should be kept as far as they can contribute to derive relevant information and pruned afterwards. In the fire alarm example, a detection of smoke needs to be kept for a specified time period. If a relevant sensor detects a high temperature during this period, a fire alarm is generated. The record of the detected smoke is disregarded afterwards.

On-demand processing is the modeling and management of data in different memory profiles such as short, episodic and semantic memories [Wood et al., 2011, Wrede, 2009, S. Wrede, M. Hanheide et al., 2004, Stachowicz and Kruijff, 2012]. Memory profiles are accessed and processed when required. An example is to calculate the position of an object in the world coordination frame from its relative position to the camera. This task requires querying the state of robot's coordination frames at the time of the observation. Due to the object recognition processing time, the state of robot's coordination frame in the past is to be queried. This requires keeping the information about the state of robot's coordination frames for some period of time. In addition, a query about the

robot's state at a time should be answered by interpolating from discrete observations of the state. This example high-lights the following on-demand processing requirements.

1. Memorizing: data should be recorded selectively to avoid overloading the memory.
2. Forgetting: outdated data should be pruned to bound the amount of recorded data in memory.
3. Active memory: memory should notify consumers when it is updated with relevant information. In this way, consumers can access the information at their time of convenience.
4. State-based representation: knowledge about the state of the robot's environment should be derived from discrete observations.

Incremental query evaluation is a processing model that mixes the on-demand and on-flow processing models as follows. Data is maintained in memory as in on-demand processing and is queried on-demand. The differences is that a query can be registered to remain active to incrementally update its results as the knowledge base changes. The updates are then sent asynchronously to the asker.

From the perspective of on-flow processing, an active query can be seen as a hypothetical on-flow processing query which has been evaluated from the earliest beginning over all histories of data. An active query can be also seen as a generalization of the notification functionality of active memories in the following two dimensions. First, a subscription in active memories only receives the matching events which occur after it has been registered. An active query, in contrast, is first evaluated over the history of data recorded in the knowledge base and is then updated as new data arrives. Second, an active memory notifies events of single changes of memory elements. In the *IDA* framework for example, one can subscribe to a pattern of such events. In a knowledge base in contrast, an active query considers all pieces of information, facts and rules, available in the knowledge base as a whole and reasons about it. Incremental query evaluation extends the notification functionality of active memories from raw data to query results.

Incremental query evaluation is necessary for reactivity of the robot. Often in planning and plan execution, the same queries are repeatedly performed on the robot's knowledge base. Such queries can include time consuming reasoning procedures on domain and perceived knowledge. Repeating the evaluation of these queries from scratch causes performance issues, negatively affecting the reactivity. To cope with this situation, a mechanism for an incremental evaluation of queries is required. Such a mechanism

should provide an efficient way of updating the results of queries according to changes of the knowledge base.

In addition to the functional requirements discussed for the above three models of processing, information engineering includes other requirements to support re-usability in implementation of its functionalities. The following presents a list of general requirements for supporting information engineering used as guidelines to develop this work.

1. Supporting on-flow and on-demand information processing and incremental query evaluation.
2. Logic-based knowledge representation and reasoning.
3. General processing: supporting external function calls, for instance, to support spatial reasoning.
4. Run-time reconfigurability: reconfigurability of functionalities at run-time.
5. Handling asynchronicity: dealing with delayed and out-of-order reception of sensory data.
6. High-level syntax: high-level syntax to support the implementation of functionalities.
7. Efficient implementation: efficient implementation of functionalities.
8. Interoperability: being framework-independent.
9. Distributed processing: distributing functionalities for modularity and efficiency.
10. Formal Semantics: clear semantics of functionalities.

### 1.2.5 Summary of State-of-the-Art

The on-flow processing support of the widely used and open-source robotic frameworks such as *ROS* and *YARP* [Metta et al., 2006] are limited to topic-based publish-subscribe communication. Topics are forms of communication channels directing messages from the publishers to subscribers. Publishers and subscribers can be added and removed at runtime. While publish-subscribe provides a basic infrastructure to facilitate the flows of data among components, the processing of data is entirely left to the components. What is missing is the support of processing, reasoning about and communication of data based on its content. Recent research on AI-based and cognitive robotic frameworks acknowledges the need for the framework-level support of on-flow processing. In

particular, the *IDA* and *DyKnow* frameworks provide some support for on-flow processing. These frameworks are however not available to the community as open-source.

The on-flow support of *IDA* is limited to content-based filtering of data. *XPATH* queries are flexible to select and transform messages based on the information they contain and the mechanism has been used in the *Filtering, Transformation and Selection* architectural pattern to increase re-usability of robot software at framework-level [Lütkebohle, 2009]. The filtering mechanism is however mostly limited to single messages. The definition of filters does not allow to correlate, aggregate and reason about various messages to extract new information.

The recent developments of *DyKnow* and, in particular, its support for *C-SPARQL* [Barbieri et al., 2010] queries enable the execution of expressive queries on flows of sensory data. Flows of data can be correlated and aggregated and ontological reasoning on background knowledge is supported. This clearly provides an advantage over existing systems. *C-SPARQL* however does not support the expression of qualitative temporal relations among data or the filtering of data patterns based on their durations. Such capabilities are desirable, if not necessary, to capture complex data patterns [Anicic et al., 2011]. In addition, on-flow processing in *DyKnow* requires semantic annotation of flows of data. Such semantic annotation is not provided, for instance, in *ROS* software widely used by the community, inducing programming overhead.

Robotic Knowledge management systems are capable of representing and reasoning on knowledge. However, they fall short in real-time processing of flows of sensory data for extraction of knowledge in dynamic environments. They process the knowledge on-demand based on the query-response model of interaction. Upon receiving a query, they perform the requested reasoning task and respond with the results. Consequently, these systems are not suitable for processing the flows of sensory data to timely detect and respond to situations of the environment. To detect a situation, an on-demand processing system needs to be frequently queried for that situation after each update of the knowledge. This approach does not scale up in practice and timely processing of data flows requires on-flow processing.

Existing knowledge management systems also lack support for efficient management and querying of histories of data and implementing event-based notifications about changes of the knowledge base. Active memories support pruning of outdated data by implementation of forgetting mechanisms. They also notify interested components when memory is updated. However, a subscription is typically limited to a single data item, filtered by its type and content and does not include information that can be derived from the whole data in memory. In summary, existing on-demand processing systems support either logical reasoning or active memories, but not both. Moreover, no system takes

into account the asynchronicity in availability of data to synchronize queries on state of the environment.

*ORO* is a logic-based knowledge management system that supports active memory functionalities such as forgetting and notification, but its support for the following on-demand processing requirements are limited. First, *ORO* does not support selective memorizing of data. All input data is recorded. Second, forgetting is limited to fixed memory profiles. It is not possible to specify time and count-based forgetting policies based on types of data. Third, due to the open world assumption, representing and reasoning about dynamics of the robot's environment is difficult in *ORO*.

*ORO* allows subscribing to events to receive notifications when triggering conditions of events become true. Triggering conditions are queries evaluated against the knowledge base. Every time the *ORO* knowledge base is altered, the *Pellet* reasoning engine of *ORO* re-classifies the whole knowledge base. Once the re-classification is completed, *ORO* tests the query of each active event. *Pellet* supports two limited forms of incremental reasoning: incremental classification and incremental consistency checking.[14] Incremental classification is used to incrementally update classification results when the class hierarchy changes. Using incremental classification, queries should be limited about classes and cannot be about instances. Incremental consistency checking supports only the addition or removal of instances, but not changes of the class and property axioms. In summary, *ORO* does not support incremental query evaluation in general and, for instance, when ontologies contain rules. For example, *ORO* uses semantic web rule language (*SWRL*[15]) [Horrocks et al., 2004] for rule-based reasoning. Such rules have to be evaluated from scratch when the knowledge base is altered.

Both *on-flow* and *on-demand* processing of sensory data are necessary for a timely extraction and dissemination of information in robot software, but current systems often support one or the other. The following situations illustrate the need to combine on-flow and on-demand processing. First, on-flow processing is needed for transforming data to a compact and suitable representation before recording it in memory. Second, a simpler and more efficient implementation of some on-flow processing tasks can be achieved by mixing on-flow pattern recognition with on-demand querying of data in memory. Third, active memories generate events when the contents of their memories change. It is desirable that a consumer is able to subscribe to notification when a pattern of such changes occurs [Wrede, 2009]. This requires an on-flow processing mechanism to process the memory events to detect relevant patterns of memory updates. Fourth, a mix of

[14] Pellet FAQ, accessed on April 10th, 2015 (https://github.com/Complexible/pellet/wiki/FAQ#incremental-reasoning)

[15] http://www.w3.org/Submission/SWRL/

on-demand and on-flow processing is necessary to support incremental query evaluation which is a generalization of the event-based notification mechanism of active memories.

Agent programming languages do not support event-driven and incremental reasoning on their input data. Therefore, the sensory input processing support of these languages is not suitable for on-flow processing of data [Ziafati et al., 2013b]. The lack of on-flow processing support reduces the reactivity and limits the application of these languages in robotics [Ziafati et al., 2013b,a]. In addition, these languages provide preliminary support for knowledge management and querying and do not provide high-level language operators for an efficient implementation of on-demand processing functionalities such as memorizing, forgetting and state-base knowledge representation.

Another concern about the application of these languages in robotics is performance issues caused by the repetition of queries on knowledge base. An approach to increase performance is to cache query results [Alechina et al., 2013]. By caching, a query is re-evaluated only if the knowledge base has been updated with relevant facts. To implement such a caching mechanism when the agent and the knowledge base components are separated, active memory notification is required to inform the agent program about the changes of the knowledge base, but such mechanism is not provided. In addition, while such a caching mechanism improves performance, cached results are invalidated when the knowledge base is updated with relevant facts. Consequently, an incremental query evaluation mechanism can improve the performance over existing approaches by propagating the changes of the knowledge base to results of queries instead of removing the cached results and re-evaluating the queries from scratch.

## 1.3 Research Questions

The overall question of this thesis is how to provide a language support for robotic information engineering that is timely processing, management and querying of asynchronous and discrete flows of sensory information to extract knowledge of the environment. In the following, this question is divided into four questions. In addition, we consider the application of agent programming languages in robotics and, in particular, their information engineering requirements as our fifth question.

The first question is how to support on-flow processing of data. A language for on-flow processing should be able to represent complex patterns of data, including temporal and logical relations and transform them into new data. General processing of data should be supported as well as interfacing with other languages to import external processing functionalities such as spatial reasoning. The syntax of language should be elegant

to support an easy and compact implementation at a suitable level of abstraction. It should also support specifying complex patterns in terms of simpler ones built on top of each other which could be re-used and easy to understand. The language should have a declarative syntax and clear semantics to support the correct implementation of functionalities and its execution engine should take the asynchronicity of data into account. Finally, event-driven and (semi) real-time processing of data requires an efficient incremental processing strategy and suitable memory management mechanisms.

The second question is how to support on-demand processing of data. An extensive study of robotics knowledge management requirements high-lights the dominant advantages of logic-based systems [Lemaignan, 2012]. The question is how to address the limitation of existing systems with regard to selecting, managing, querying and synchronizing the relevant parts of flows of sensory data in the knowledge base and support active memory notification. Language support is required to enable the definition of high-level policies for selective recording of data and pruning outdated data. It is also required to facilitate the state-based representation of data built upon discrete observations of the environment (i.e. events). An efficient management and querying of histories of data requires their underlying management in suitable data structures and using indexing mechanisms. Active memory notification requires support for generation and management of events. Finally, language support is needed to synchronize queries on the state of the environment, built upon events, asynchronously received from the perception components.

The third question is how to support incremental evaluation of queries on the knowledge base. Language support is needed to be able to register queries as active queries that are evaluated on the current knowledge in the knowledge base and their results are updated in real-time according to changes of the knowledge base. The question is how to provide an efficient mix of top-down (i.e. query-driven) and bottom-up (i.e. data-driven) query evaluation strategies deployed in on-demand and on-flow processing models. Given a query, a top-down evaluation strategy examines the relevant rules and facts in the knowledge base to find answers of the query. The advantage is that the search strategy is guided by the type of the query and its bound variables. In bottom-up approaches, queries of interest are usually known in advance and are part of the knowledge base. The knowledge base then explicitly stores all relevant results that can be derived from the given facts and rules in the knowledge base. The advantages of these approaches is their use of data-driven evaluation mechanisms to efficiently update knowledge base, including the results of queries of interests, as facts are added or removed. The question is how to efficiently evaluate queries in a top-down manner and efficiently update their results in a bottom-up manner only taking into account the queries which are active at the time.

The fourth question is how to uniformly support the on-flow, on-demand and incremental query evaluation functionalities in a language. In particular, logic-based approaches are of interest for modeling, representing and reasoning on knowledge. The question is how to develop a logic-based language that can process, manage and query the flows of data in real-time to provide a more comprehensive support of information engineering with respect to the current robotic knowledge bases.

The fifth question of this thesis is about the application of agent programming languages in robotics. We investigate what the plan execution control requirements of these language in robotics are and how to support the requirements. In particular, we are interested in the relation between such requirements and the information engineering requirements of these languages. We ask the question of whether and to what extent the information engineering techniques developed in this thesis address the information engineering requirements of these languages and support the real-time and event-driven execution of plans to apply these languages in robotics.

## 1.4 Methodology

Identifying general robotic information engineering requirements is challenging due to the wide range of robotic systems and applications. The main approach followed in this thesis is to generalize from the requirements in various research related to robotic sensory data processing and management. This includes knowledge processing and event-based frameworks [Heintz et al., 2010b, Wrede, 2009, Hawes and Hanheide, 2010], active memories for sensory data fusion [Bauckhage et al., 2008, S. Wrede, M. Hanheide et al., 2004, Hawes and Wyatt, 2010, Hawes et al., 2008], knowledge management systems [Tenorth and Beetz, 2009, 2012, Lemaignan et al., 2011, Lemaignan, 2012] and on-flow processing systems [Lütkebohle, 2009, Heintz et al., 2010b, 2013, Heintz and Leng, 2013, de Leng and Heintz, 2014, Heintz, 2013, Ranathunga et al., 2012, Pecora et al., 2012, Sabri et al., 2011, Buford et al., 2006].

We take the same approach to identify the plan representation and execution requirements of agent programming languages in robotics. We generalize from the functional capabilities of existing robotic plan representation and execution control languages including TDL [Simmons and Apfelbaum, 1998], PLEXIL [Tara and Vandi, 2006], APEX Freed [1998], SMARTTCL [Steck and Schlegel, 2010], PRS [Georgeff and Lansky, 1987] and PRS-lite [Myers, 1996].

Software development is inherently of high importance for this thesis work due to the following reasons. First, the aim of this work is the support of timely processing, management and querying of information. This includes developing a language that provides a high-level syntax and clear semantics to specify information engineering functionalities. The main key to usefulness of such a language is its efficiency in execution of the processing, management and querying functionalities. For example, while it is clear that on-flow processing requires an event-driven processing model, the efficiency of a specific approach and its implementation needs to be evaluated in practice.

Second, there has been already a large effort in the development of robotic frameworks. In particular, *ROS* has emerged as the de-facto standard robotic framework and it is being widely used and further developed by the community. *ROS* and other frameworks provide a basic support for synchronous and asynchronous communications among components. We assume the support of such communication mechanisms as given in existing systems and choose to build our software on top of it. Integration with existing software such as *ROS* is important not to re-invent the wheel and, more importantly, to make the developed software accessible and promote its use by the community. In this regard, the developed software should be easy to integrate with existing robotic frameworks and the efficiency of the integration, and in particular, the data format conversion necessary for the integration requires an experimental evaluation.

Due to these reasons, we put a great emphasis on software development and prototyping in this thesis. The implementation of algorithms and demo applications serves us to develop, test and enhance the usability of our approach. In addition, it serves as an important goal of this thesis to enhance the robotic software engineering experience by providing corresponding tools as open-source for the community. In the rest of this section, we describe the particular approaches taken to answer the research questions of this thesis.

### 1.4.1 On-Flow Processing

We consider the component interactions, plan execution and monitoring, anchoring and situation recognition as four main situations where on-flow processing of data is useful in robotics. From an analysis and discussion of these situations, we derive the main on-flow processing requirements.

To address the requirements, we look into information-flow processing systems. These systems provide efficient languages for processing large volume of flow of information. There are two classes of such systems: *Data Stream Management Systems* (*DSMSs*) and *Complex Event Processing Systems* (*CEPSs*) Cugola and Margara [2012]. *CEPSs* are

of more interest due to their expressiveness in presenting complex temporal relations between events to detect complex events. Among the existing *CEPSs*, we propose to integrate and extend the *ETALIS* language for event-processing (*ELE*)[16] [Anicic et al., 2012, 2010, Anicic, 2011] for on-flow processing of sensory information in robotics.

#### 1.4.1.1 *ETALIS* Language for Event-Processing (*ELE*)

*ELE* is a language for detecting and processing complex events in flows of events. An event is a time-stamped piece of information. An *ELE* program consists of a set of event rules. The body of an event rule specifies a pattern of events to be detected. The head of the rule specifies a new event to be generated for each set of events which match the pattern specified by the body. The head and body of a rule share variables. In this way, information is passed from the events matching the body to the corresponding event generated by the head. For example, the rule "a(X) ← b(X) SEQ c(X)" is informally read as follows. For each pair of events of type $b$ and $c$, where the event of type $b$ occurs in advance and they have the same arguments, an event of type $a$ is generated with the same argument. In *ELE*, events generated by the rules can themselves match the body of other rules contributing to generation of other events. This allows to specify complex events in terms of simpler ones.

*ELE* is an expressive language allowing the representation of all possible thirteen temporal relations among time interval occurrence times of two events (e.g. an event occurring before the other, they occur at the same time, etc). It can also represent non-occurrence of an event between the occurrence of two other events. In addition, *ELE* allows to set a time limit within which a set of events matching a body of an event rule should occur. For instance, the rule "a(X) ← (b(X) SEQ c(X)).5sec" specifies that a pair of events of type $b$ and $c$ generates a corresponding event of type $a$, if these events occur in the specified sequence order and they occur within a time period of five seconds.

In addition to specify temporal relations among events, *ELE* allows also reasoning over background knowledge as follows. Background knowledge is encoded as a set of logical rules and can be reasoned about when testing whether a set of patterns match the body of an event rule. To this end, an event rule includes a logical query in its body. When a set of events match the body, this query is instantiated according to the contents of those events (i.e. some of its variables are bound to some values). The query is then evaluated over the background knowledge and those events are considered as being matched to the body only if the query has an answer. The ability to reason on background knowledge is a distinguishing feature of *ELE* comparing to other *CEPSs*. *ELE* bridges the gap

[16] Etalis (Event TrAnsaction Logic Inference System) provides also the Event Processing SPARQL language (EP-SPARQL) for event processing in Semantic Web applications.

between logical query-response approaches for event representation and recognition and incremental but non-logical approaches for real-time processing of information flows.

*ELE* offers a logic-based *CEPS* with an event-driven, incremental and efficient execution model. The execution model of *ELE* enables the effective detection of complex events at run-time following the semantics of the language. Every time an event occurs, the system updates its knowledge base, encoding which events have already happened and which ones are missing for the completion of complex events. A complex event is detected as soon as the last event required for its completion occurs. *ELE* uses *Prolog* as its execution system. A typical *Prolog* program usually follows the query-response execution model where queries are asked and answers are computed. To provide an incremental and event-driven execution model, *ELE* programs are transformed into a set of rules called goal-driven backward chaining rules. The transformation is such that, first, the final program implements the semantics of *ELE*, informally described above, second, the computations are driven by new events received by the *ELE* execution systems, and third, complex events are efficiently detected in an incremental manner. Using such a model and particular data structures, *ELE* achieves a performance of comparable to state-of-the-art non logic-based *CEPSs*.

*ELE* comes with a *Java* wrapper providing *Java* class templates to implement the necessary communications with other software components using network sockets. Using this wrapper, the system is connected to a fixed set of provider components, receiving events they generate. It is connected also to set of consumer components, sending certain types of events to those components. We extend *ELE* and its execution system in three ways. First, the Java wrapper is replaced with a much lighter C++ wrapper, considerably improving the performance. Second, we provide a tool for an automatic conversion of *ROS* messages to *ELE* events and vice versa. Third, we provide a run-time subscription mechanism. Using this mechanism and corresponding interfaces, *ELE* can be (un)subscribed to new *ROS* topics at runtime. In addition, *ROS* components can (un)subscribe to *ELE* at runtime for events of their interests, filtered by their contents.

### 1.4.2 On-Demand Processing

Logic-based approaches for on-demand information processing in robotics are dominant [Lemaignan, 2012]. We use *Prolog* as the underlying knowledge representation and reasoning system due to its closed-world assumption and supporting a form of non-monotonicity by the *negation as failure* inference rule. These characteristics bring *Prolog* some practical advantages for a more compact knowledge representation and facilitating reasoning about changes comparing to, for instance, existing description logic reasoners

such as the *Pellet*[17] reasoner [Sirin et al., 2007]. In addition, the query-response execution model of *Prolog* is naturally more suitable for on-demand processing, comparing to *DL* reasoners that re-classify the whole knowledge base on every update. Moreover, the availability of software libraries to interface *Prolog*, for instance, with the *C++* language makes it easy to integrate additional reasoning capabilities or information from external sources in on-demand processing. Finally, the *ELE* language is parsed and executed by *Prolog*. Having *Prolog*-based languages for both on-demand and on-flow processing enables developing an integrated system on top for robotic information engineering.

In *Prolog* syntax, a *term* is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a functor symbol and $t_1, \ldots, t_n$ are constants, variables or terms. A term is *ground* if it contains no variables. A *Horn* clause is of the form $a_1 \wedge \ldots \wedge a_n \rightarrow a$, where $a$ is a term called the *Head* of the clause, and $a_1, \ldots, a_n$ is called the *Body* where $a_i$ are terms. In *Prolog* syntax, the body can also include negation of terms. $a \leftarrow true$ is called a *fact* and usually written as $a$. A *Prolog* program $P$ is a finite set of *Horn* clauses. One executes a logic program by asking it a query of the form $b_1 \wedge \ldots \wedge b_n$ where $b_i$ is a term. *Prolog* employs the *SLDNF* resolution method [Apt and van Emden, 1982], a depth-first search strategy, to determine whether or not a query follows from the program. Given a goal, *SLDNF* tries to prove the goal using the rules and facts of the program. A goal is proved if there is a variable substitution by applying which the goal matches a fact, or matches the head of a rule and the goals in body of the rule can be proved from left to right. A variable substitution is a mapping from a set of variables to a set of terms. Goals are resolved by trying the facts and rules in the order they appear in the program. A query may result in a substitution of free variables.

We extend the *Prolog* language with domain-specific language operators and constructs to support on-demand processing functionalities such as memorizing, forgetting and state-based knowledge representation. To this end, we aim for simple and minimal language extensions to enable high-level and expressive specifications of a general set of such functionalities. In addition, we take the efficiency requirement into design of such operators and constructs such that indexing mechanisms can be used for their efficient underlying implementations. To deal with asynchronous reception of events by the knowledge base, the semantics of the proposed functionalities take into account the occurrence times of events, as opposed to the time the events are received by the knowledge base. Using such semantics, we investigate mechanisms to synchronize the queries about the state of the environment, built upon the asynchronous events.

The result is a *Prolog*-based language called *Synchronized Logical Reasoning* language (i.e. *SLR*). It supports selective recording of data in memory items that are assigned

[17] http://clarkparsia.com/pellet/

memory profiles, efficiently implementing forgetting mechanisms. In addition, it supports active memory notification where events are generated whenever a new event is recorded or removed from the knowledge base. Furthermore, *SLR* provides two simple operators to efficiently interpolate or extrapolate the state of a domain, observed by discrete events and supports automatic synchronization of queries.

### 1.4.3 Incremental Query Evaluation

Choosing *Prolog* as our underlying knowledge management system, we investigate an approach to support an incremental evaluation of logic programming queries in *Prolog*. To this end, we build on the idea of goal-directed backward chaining rules used in the execution model of *ELE* for an incremental and event-driven detection of complex events.

The syntax of *ELE* is easily mapped to *Prolog* syntax. For example, the event rule "a(X) $\leftarrow$ b(X) SEQ c(X)" is mapped to the "a(X) :- b(X) $\wedge$ c(X)" *Prolog* rule.[18] However, the operational semantics of *ELE* and *Prolog* are different. In an *ELE* program, the head of each rule specifies an event to be detected based on the events specified in the rule's body. In a *Prolog* program, a rule specifies that a goal matching its head can be proven, if all sub-goals in its body can be proven. In other words, *Prolog* is goal-driven and find answers for a given query, but *ELE* is event-driven and find all answers that can be derived by the rules in its program. Moreover, new facts are stored in *Prolog* to be used in future to derive answers for the queries, but *ELE* uses new events to derive complex events and disregards them afterwards. Consequently, while *ELE* provides an event-driven and incremental execution model, it has the following limitations which need to be addressed for incremental query evaluation.

*ELE* derives all facts (i.e. events) that can be derived by every rule in its program, but incremental query evaluation requires to only derive the facts that match the given active queries, queries whose results are of interest at the time. *ELE* rules can be added or removed at runtime, but this ability is not enough for implementation of active queries due to the following reasons. Given a set of active queries, there needs to be an automatic way of choosing the necessary set of rules to be used for deriving answers to these queries which is not supported. More importantly, when an *ELE* rule is added, it only considers the facts which are added to the knowledge base afterwards and the facts already in the knowledge base are disregarded. In addition, *ELE* rules are amenable to easily fall into infinite loops. For instance, if we have the two rules, " a(X) $\leftarrow$ b(X)" and "b(X) $\leftarrow$ a(X)", and we provide $a(1)$ as input event, the *ELE* execution system falls into an infinite loop and never stops.

[18] There should be an additional clause in the rule to compare the time stamps of the events to check for their occurrence in the specified sequence order, omitted for brevity.

We extend the *ELE* approach in order to have only those rules activated that are necessary. A rule should be activated, if the system is currently evaluating a query and the answers generated by the rule are relevant to finding answers for the query. In addition, we adapt the approach such that when a query is registered, all intermediate goals are efficiently generated as if the query was registered before the addition of any fact to the knowledge base. Furthermore, we borrow the idea of caching sub-goals resulted from research on tabled logic programs [Swift and Warren, 2010, Saha and Ramakrishnan, 2006a, 2003]. In tabling, results generated for a sub-goal are cached and next time the sub-goal is encountered, the cached results are used instead of re-evaluating the sub-goal. By tabling, the performance is improved. Moreover, the problem of falling into an infinite loop for rules with left recursion is prevented.

### 1.4.4 Mix of Approaches

Our approaches to support on-flow, on-demand and incremental query evaluation functionalities are all based on logic programming and are implemented and executed by the *SWI-Prolog* system. Consequently, they can all be combined to provide an integrated system to provide a comprehensive support for necessary processing models required in information engineering.

The *ELE* language provides a built-in support to execute *Prolog* queries. This facilitates to perform *SLR* queries in *ELE* as *SLR* queries are in the *Prolog* query format and are executed by the same *Prolog* execution system that executes *ELE*. Therefore the basic interface from *ELE* to *SLR* is already in place. The interface from *SLR* to *ELE* is event-driven. We feed the *SLR* events, generated due to changes of the *SLR* knowledge base, to the *ELE* execution system, considered as *ELE* typical input events which can be captured to derive complex events. In addition, events generated by *ELE* are fed as input events to *SLR*, the history of which can be maintained.

By integration of *SLR* and the extension of *ELE* with runtime subscription, we develop the *Retalis* language (*ETALIS* for Robotics) to support the implementation of both on-flow and on-demand information engineering functionalities in one program. *Retalis* is open-source software, released as a *ROS* package.[19] Our approach for incremental evaluation of queries has not yet been included in the current release of *Retalis*. The integration requires the development of necessary interfaces in *ROS* allowing components to register queries to asynchronously receive updates on query results which have been left for future work.

[19] http://wiki.ros.org/retalis

### 1.4.5 Plan Representation and Execution in Agent Programming

To address the plan representation and execution requirements of agent programming languages, we opt to build upon the *PLEXIL*[Verma et al., 2005, Gilles Dowek, César Muñoz and Pasareanu, 2010] plan execution language developed at *NASA* due to the following reasons. *PLEXIL* offers a simple structure for plan representation, a hierarchy of nodes with few syntactic constructs, but it is one of the most expressive plan execution languages unifying many of the existing ones. Moreover, *PLEXIL* has formal semantics which allows for the formal study of various types of determinism of plan execution. In addition, the operational semantics of *PLEXIL* is presented in a modular way at various levels of plan execution easing the formal study and modification of the language. Finally, the languages has been successfully used in various robotic applications.

To address the requirements, we develop the *RobAPL* language. *RobAPL* adapts the *PLEXIL* syntax and semantics to be integrated in BDI-based agent programming languages for representing and executing plans. We introduce new execution nodes for querying and manipulating the agent's beliefs and goals and present an operational semantics for *PLEXIL*-like plan execution in the BDI architecture. Moreover, *PLEXIL* is extended to support pausing, resuming and pre-empting plans, performing clean-up and wind-down activities when pausing, resuming, pre-empting or aborting plans, and coordinating the parallel execution of plans over shared resources.

## 1.5 Thesis Layout

This document is organized as follows. Chapter 2 presents a running example and gives an architectural overview of *Retalis*. Chapter 3 discusses on-flow processing requirements and introduces information flow processing systems and, in particular, the *ELE* language as suitable technologies for on-flow processing of robotic sensory data. The *ELE* language is presented and extended with a dynamic subscription mechanism to support run-time content-based flow of information in robot software. Chapter 4 discusses on-demand processing requirements and extends *Prolog* into the *SLR* language for robotic on-demand processing. Chapter 5 provides an empirical evaluation of the on-flow and on-demand functionalities of *Retalis* by implementing a demo application for the *NAO* robot. Chapter 6 develops an approach for incremental evaluation of definite logic program queries and evaluates its performance. Chapter 7 presents the *RobAPL* language and discusses the role of information engineering and the approach taken in this thesis in autonomous robot programming using agent programming languages. Finally, Chapter 8 presents a summary and future work.

# Chapter 2

# *Retalis* Language for Robotic Information Engineering

This chapter answers the question of how to uniformly support various models of information processing in a robotic information engineering language. In the Introduction chapter, we discussed that logic-based approaches are of interest for modeling, representing and reasoning about knowledge, but existing systems fall short in efficiently processing, managing and querying flows of sensory information. This chapter presents an architectural overview of the *Retalis* language developed in this thesis to address such shortcomings. *Retalis* is open source software which has been integrated in *ROS*.[1]

The remainder of this chapter is organized as follows. We first present a running example used throughout out this thesis to illustrate the on-flow and on-demand information engineering supports of *Retalis*. We then discuss how sensory information is represented in *Retalis* and is processed and managed by *ELE* and *SLR* languages used in *Retalis* to develop the *Information Engineering Components* (*IECs*) of autonomous robots. The API of *Retalis* to develop synchronous and asynchronous interactions among *IECs* and other robot's software components are presented and the integration with *ROS* is discussed.

## 2.1 Running Example

This section presents an example to illustrate the concepts and functionalities of *Retalis*. A robot is situated in a dynamic environment informing a person about the objects around it. The environment is described by a set of entities $e_1, e_2, ....$ These include the moving *base* of the robot, the pan-tilt 3D camera *cam* of the robot mounted on the *base*,

[1] http://wiki.ros.org/retalis

a set of tables $table_1, table_2, ...$, a set of objects $o_1, o_2, ...$, a set of people $f_1, f_2, ...$, a set of attributes and a reference coordination frame *rcf*.

Figure 2.1 presents the robot's software components and their interactions. This figure should be read as follows. Directed arrows visualize asynchronous flow of data and two-way arrows represents request-response service calls.

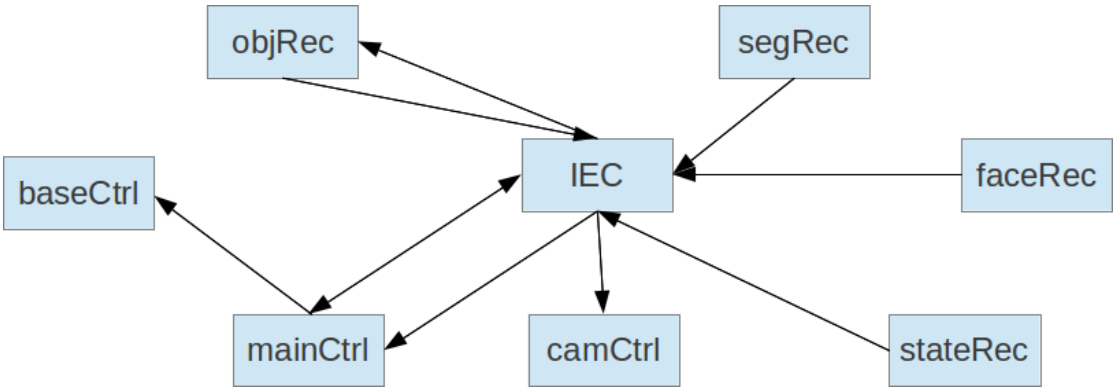


FIGURE 2.1: Robot's software components

The robot software includes the following components. Asynchronous communications among these components are in the form of events. An event is a time-stamped piece of data formally defined in Section 3.3.

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

*segRec* component: uses a real-time algorithm to process images from the camera into 3D point cloud data segments corresponding to individual objects. Such an algorithm is presented by Uckermann et al. [Ückermann et al., 2012]. The *segRec* component outputs $seg(o_i, c_j, p_k, l_g, pcl_h)^t$ events. Such an event represents the recognition of object $o_i$, with color $c_j$, with probability $p_k$, with relative position $l_g$ to the *cam*, with the 3D point cloud data segment $pcl_h$ recognized from a picture taken at time $t$. For events of the recognition of the same object segment over time, a unique identifier $o_i$ is assigned using an anchoring and data association algorithm. Such an algorithm is presented by Elfring et al. [Elfring et al., 2012].

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

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

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

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

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

## 2.2 Architectural Overview

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

*Retalis* represents and manipulates data as events. Events are time-stamped discrete pieces of data whose syntax is the same as *Prolog* ground terms [Clocksin and Mellish, 2003, Lloyd, 1984a]. Events contain perceptual information such as a robot's position at a time or recognized objects in a picture. The meaning of events is domain-specific. The time-stamp of an event is a time point or a time interval referring to the occurrence time of the event. Events are time-stamped by the components generating them. [2] For example, the event *face('Neda',70)*[28] could mean a recognition of Neda's face with 70%

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

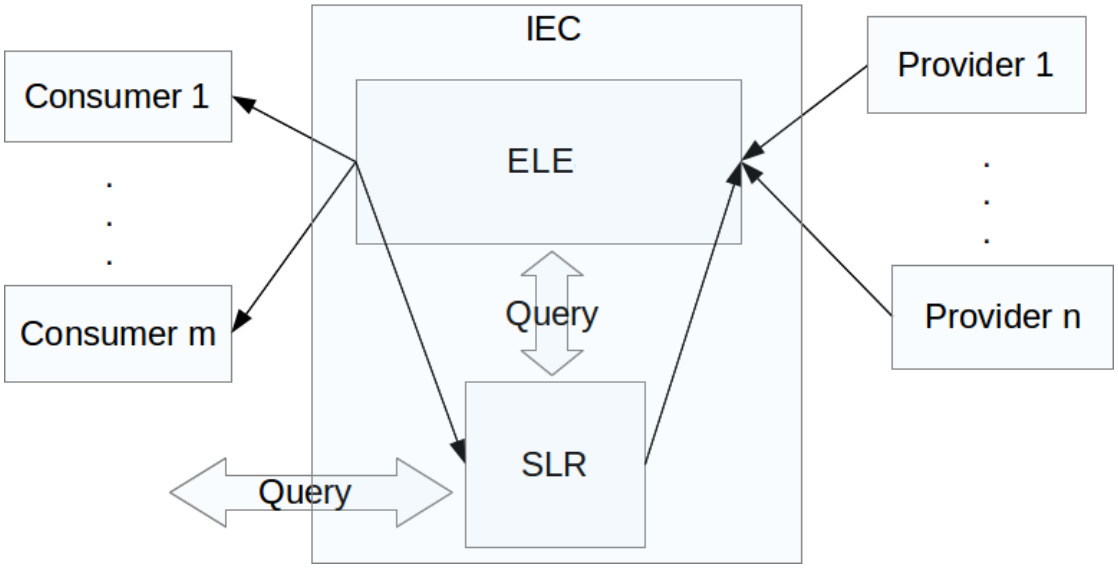


FIGURE 2.2: *IEC* architecture

confidence in a picture taken at time 28 and the event *observed('Neda')*$^{\langle 28,49\rangle}$ could mean a frequent recognition of Neda's face in pictures taken during time interval [28,49]. An event containing information from processing of sensory data is usually time-stamped with the time at which the sensory data is acquired. This is usually different from the time point when the processing of the data is finished. A composite event generated from an occurrence of a pattern of other events is time-stamped based on the occurrence times of its composing events.

*Retalis* comprises two logic-based languages. The *ELE* language [Anicic et al., 2012, 2010, Anicic, 2011] supports on-flow processing and the *SLR* language [Ziafati et al., 2014] supports on-demand processing of data. In the *Retalis* program of an *IEC*, *ELE* generates composite events by detecting event patterns of interest in the input flow of events to the *IEC*. *SLR* is used to implement a knowledge base maintaining the history of some events. The knowledge base contains domain knowledge, including rules to reason about the recorded history. The flow of events processed by the IEC includes its input events and the composite events it generates. This means that composite events can in turn be used to detect other events. The robot software presented in Section 2.1 includes one *IEC* component. Robot software can include a number of *IEC* components in order to modularize different information engineering tasks and to use distributed and parallel computing resources.

Figure 2.2 depicts the architecture of an *IEC*, including its logical components implemented in *Retalis*. This figure must be read as follows. Directed arrows visualize asynchronous flow of events. Two-way arrows represent queries to *SLR* by *ELE* and external

components.

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

Synchronous interactions between the *IEC* and other components are as follows. Components can query the domain knowledge and history of events in the *SLR* knowledge base. *Retalis* provides a request-response service to query *SLR*. *SLR* is a *Prolog*-based language, presented in Section 4.2 . The evaluation of a *SLR* query determines whether the query can be inferred from the knowledge base. The query evaluation may result in a variable substitution. The *IEC* can also access the functionalities of other software libraries or components. Function calls are supported both when answering queries and detecting composite events. To integrate external functionalities in *Retalis*, the corresponding software libraries should be interfaced with *Prolog*.

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

An *ELE* program, described in Section 3.3, contains two types of rules. The rules that include the ← symbol are event rules, specifying patterns of events to derive new events. The rules that include the *:-* symbol are static rules, constituting a *Prolog* program. The specification of the pattern of events in an event rule can include a query to the *Prolog* program defined by the static rules. *Retalis* programs are similar to *ELE* programs. The main difference is that the static rules in *Retalis* are *SLR* rules, constituting a *SLR* program which can be queries from the event rules.

Listing 2.1 presents an example of how *ELE* and *SLR* are used together in a *Retalis* program. This program records the position of the object segment $o_1$ whenever the position is changed by more than a meter. This program is read as follows. Capital letters represent variables. The body of the first and third rules are executed when the program is initialized. *c_mem($m_1$,loc($o_1$,L),∞,∞)* is a *SLR* clause creating memory $m_1$ recording the history of *loc($o_1$,L)*$^T$ events. The second rule is an *ELE* clause querying *SLR*, as written in its *WHERE* clause. For each *seg($o_1$,C,P,L,PCL)*$^T$ input event, the *prev* clause queries memory container $m_1$ for the last position of $o_1$ before time $T$. If the position has changed by more than a meter, the corresponding *loc($o_1$,L)* event is generated and recorded in memory $m_1$. In addition, consumer *moving_objects* is notified by the corresponding event *obj($o_1$)*$^T$. This is specified by the third rule, which is read as follows. The subscription $s_1$ subscribes consumer *moving_objects* to *loc(O,L)* events with the output template *obj(O)* from time 0. Details of the *ELE* and *SLR* languages are given in Sections 3.3 and 4.2.

```
1  on_program_start :- c_mem(m1, loc(o1,L)^T, ∞, ∞).
2
3  loc(o1,L)^T <- seg(o1,C,P,L,PCL)^T
4                 WHERE(
5                       prev(m1, loc(o1,L_prev)^T_prev, T)
6                       dist(L,L_prev,D),
7                       D > 1
8                      ).
9
10 on_program_start :- sub(s1, moving_objects, loc(O,L), obj(O), 0).
```

Listing 2.1: *Retalis* Program Example

A *Retalis* program is parsed and executed by a *Prolog* execution system and is provided a *C++* interface for communication with external components. This makes the *Retalis* language framework-independent, because its core depends only on a *Prolog* execution system. We use *SWI-Prolog*[3] [Wielemaker et al., 2012] as the *Retalis* execution system and use the *SWI-Prolog C++ interface*[4] to interface the *SWI-Prolog* with *C++*. *Retalis* can be interfaced with existing robotic frameworks mapping its synchronous and asynchronous interfaces to their service-based and publish-subscribe communication mechanisms.

[3] http://www.swi-prolog.org
[4] http://www.swi-prolog.org/pldoc/package/pl2cpp.html

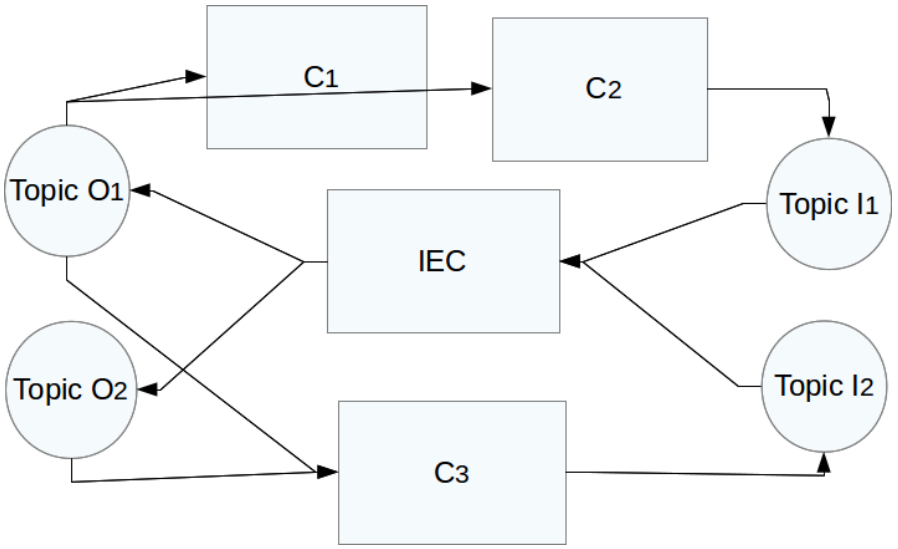


FIGURE 2.3: An *IEC* in *ROS* architecture

We have developed an interface to integration *Retalis* with the *ROS* framework [Quigley et al., 2009], the current de-facto standard in open-source robotics. In the *ROS* architecture, each *IEC* is a *ROS* component[5] [Quigley et al., 2009]. Asynchronous and synchronous communications in *ROS* are realized using topics and services, respectively. By subscribing to a topic, a component receives the messages other components publish on that topic. A component invokes a service by sending a request message and receiving a response message.

Figure 2.3 presents an *IEC* in a *ROS* architecture. *IEC* is subscribed to Topics $I_1$ and $I_2$ receiving messages published by the components $C_2$ and $C_3$. *IEC* publishes events on topics $O_1$ and $O_2$ to which other components are subscribed.

To subscribe an *IEC* to a topic, the *Retalis-ROS* interface requires the name and message type of the topic. This is set in an *XML* configuration file, as in line 4-6 of Listing 2.2. The *Retalis-ROS* interface offers a number of services to reconfigure the *IEC* at runtime. These include services to subscribe the *IEC* to a topic, to un-subscribe from a topic and to subscribe a topic to events from the *IEC*. To publish an event on a *ROS* topic, the *Retalis-ROS* interface needs to know the message type of that topic. This can be set by the program, as in lines 7-9 and 10-12 of Listing 2.2, or at runtime.

The *Retalis-ROS* interface provides also services to query the *SLR* knowledge base of *IEC*. There are two types of queries, one asking for an answer to a query and another asking for all answers to a query. A request message of query services is a *SLR* query,

[5] http://wiki.ROS.org/Nodes

represented as a String value. An answer message of the single-answer query is a substitution. A substitution is a list of variable and value tuples. An answer message of the multi-answer query is a list of substitutions.

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

LISTING 2.2: *Retalis-ROS XML* configuration file

The conversion among *ROS* messages and *Retalis* events is performed automatically by the *Retalis-ROS* interface. This may be described by an example. Table 2.1, consisting of five columns, depicts five standard *ROS* message types. The first row in each column is the name of a unique message type. The other rows presents the fields of data that the message type contains. Each field of a message contains a single datum or a list of data, whose type is a basic type such as Integer, Float, String, or it is a *ROS* message type. For example, a *geometry_msgs/Point* message contains three float values and a *geometry_msgs/Pose* message has a *geometry_msgs/Point* message as its first field of data.

Listing 2.3 presents the conversion of the *geometry_msgs/PoseStamped ROS* message type to its corresponding *Retalis* event. The conversion maps each *ROS* message to a *Prolog* compound term where the functor symbol of the term is the name of the message type and its arguments are the data fields of the message. Data of basic types such as Integer and Floats are represented by their values. Strings are wrapped by single quotes represented as *Prolog* Strings. Lists of data are represented as *Prolog* lists. Time in *ROS* is a basic data type expressed by two Integer values represented in a *Retalis* event as a list of two numbers.

When converting a *ROS* message to a *Retalis* event, the event is time-stamped with the time-stamp of the header of the message. If the message does not have a header, the event is time-stamped with the system current time. When converting a *Retalis* event to

| geometry_msgs/PoseStamped | |
|---|---|
| std_msgs/Header | header |
| geometry_msgs/Pose | pose |

| std_msgs/Header | |
|---|---|
| uint32 | seq |
| time | stamp |
| string | frame_id |

| geometry_msgs/Pose | |
|---|---|
| geometry_msgs/Point | p |
| geometry_msgs/Quaternion | o |

| geometry_msgs/Point | |
|---|---|
| float64 | x |
| float64 | y |
| float64 | z |

| geometry_msgs/Quaternion | |
|---|---|
| float64 | x |
| float64 | y |
| float64 | z |
| float64 | w |

TABLE 2.1: *ROS* message examples

a *ROS* message, the time-stamp of the event is ignored. However, the *Retalis* language provides direct references to time-stamp of events. This can be used, for instance, to set the *stamp* in the *std_msgs___header(seq,stamp,frame_id)* argument of an event and hence in the header of its corresponding *ROS* message. *ROS* messages from different topics can be of the same type and need to be distinguished. Therefore, we encode topic names as main functor symbols of corresponding *Retalis* events. For example, if the event $p_n(t_1,..,t_n)^z$ is received from the topic $x$, the event is represented as $x(p_n(t_1,..,t_n))^z$.

```
1 geometry_msgs____PoseStamped(
2           std_msgs____Header(seq,stamp,frame_id),
3           geometry_msgs____Pose(
4                                 geometry_msgs____Point(x,y,z),
5                                   geometry_msgs____Quaternion(x,y,z,w)
6                                   )
7                                )^stamp
```

LISTING 2.3: *Retalis* event format corresponding to geometry_msgs/PoseStamped *ROS* message type

## 2.3 Summary

The logic programming based *Retalis* language is introduced to develop *Information Engineering Components* (*IECs*) of autonomous robots. In the *Retalis* program of an *IEC*, the *ELE* language is used to program a set of rules to process input flows of events

to the *IEC* on the fly. Complex temporal and logical patterns of events are detected, based on which new events are generated. In the same program, the *SLR* language is used specifying a knowledge base to selectively maintain and reason about the history of events and domain knowledge. The interaction between the *ELE* and *SLR* parts of a *Retalis* program is three-fold. First, *ELE* rules can query the information maintained by *SLR* while detecting patterns of events. Second, *SLR* generates events when its knowledge base is updated with new information. Such events are fed into *ELE* as its input events which can be captured to detect complex events. Third, events generated by *ELE* are fed to *SLR* as its input events, the history of which can be maintained by *SLR*.

An *IEC* receives asynchronous events from other components and asynchronously sends the events it generates to other components based on their runtime interests. Components can also query the history of events maintained by the *IEC* and the *IEC* can also access data or services from other components on-demand. *Retalis* has been integrated in *ROS* providing user-friendly *API* to implement these synchronous and asynchronous interactions and to automatically convert between *ROS* messages and *Retalis* events. Components can also register queries to an *IEC* to asynchronously receive updates on their results as the *SLR* knowledge base of the *IEC* is updated. This mechanism however is not included in the current open-source release of the *Retalis*.

# Chapter 3

# On-Flow Information Processing

The question of this chapter is how to support on-flow processing of information. A language for on-flow processing should be able to represent complex patterns of data, including temporal and logical relations and transform them into new data. General processing of data should be supported as well as interfacing with other languages to import external processing functionalities such as spatial reasoning. The syntax of language should be elegant to support an easy and compact implementation at a suitable level of abstraction. It should also support specifying complex patterns in terms of simpler ones built on top of each other that could be re-used and easy to understand. The language should preferably have a declarative syntax and clear semantics to support the correct implementation of functionalities and its execution engine should take the asynchronicity of data into account. Finally, event-driven and real-time processing of data requires an efficient incremental processing strategy and suitable memory management mechanisms.

We discuss on-flow processing requirements of robotic information engineering and suggest that information flow processing systems [Cugola and Margara, 2012] are suitable technologies to address the requirements. Among the existing information flow processing systems, we choose the *ETALIS* event-processing language *(ELE)* [Anicic et al., 2012, 2010, Anicic, 2011] for on-flow information processing in robotics. The reason is, in particular, that *Etalis* provides a logic-based language for event-driven and efficient processing of flows of sensory information, enabling logical reasoning on domain knowledge in processing flows of information. Moreover, *Etalis* is easy to interface with other languages to integrate, for instance, spatial reasoning capabilities.

The remainder of this chapter is organized as follows. We first discuss the component interactions, plan execution and monitoring, anchoring and situation recognition as four main situations where on-flow processing of data is very useful in robotics. We then

briefly discuss the information flow processing systems and present a detailed account of the *ELE* language of *Etalis*. Afterwards, We extend *Etalis* with a subscription mechanism allowing a run-time configurable content-based filtering of flows of data. Finally, we present a detailed comparison of the *Retalis* support for on-flow processing, realized by integrating the *Etalis* system and extending it with a runtime subscription mechanism, with existing work and give a summary.

## 3.1 On-Flow Processing Requirements

On-flow processing of data is widespread in large areas of robot software. As examples, the following presents four robotic situations where on-flow processing of data is very useful.

The first situation is decoupling components interacting in robot software. This is usually supported by a publish-subscribe communication mechanism [Eugster et al., 2003] based on an indirect addressing style [Brugali and Shakhimardanov, 2010, Wrede, 2009, Quigley et al., 2009, Heintz et al., 2010b]. The publish-subscribe mechanism organizes robot software in a data-driven manner where components continuously process data generated by the other components. However, due to the limited resources of a robot, sensory data needs to be processed selectively. This requires filtering of data passed among components. Data should be filtered based on the robot's operational context, such as its focus of attention.

One way to support the filtering of data according to the robot's operational context is to write complex software components whose processes can be reconfigured at runtime. However, such a reconfiguration might not be supported by the available components. The publish-subscribe support in most existing robotic frameworks such as *ROS* [Quigley et al., 2009] and *YARP* [Metta et al., 2006] is limited to topic-based interactions. Providers publish data items on topics, which are received by subscribers to those topics. In these frameworks, a component is usually subscribed to a fixed set of topics.

More flexible and context-dependent interaction requires subscribers being able to specify their data of interest based on data patterns and policies [Wrede, 2009, Heintz et al., 2010b, Lütkebohle, 2009]. Consider a robot looking for reliable recognition of yellow objects. The object segments sent to the object recognition component should be filtered to include only the yellow and reliably recognized object segments. Another example is the selective processing of new perceptions of object segments by the object recognition

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

The second situation is anchoring [Coradeschi and Saffiotti, 2003], creating symbolic representation of objects perceived from sensory data. The symbols and the data continuously sensed about the objects should be correlated. In an anchoring process, sensory data is interpreted into a set of hypotheses about recognized objects. For example, in a traffic monitoring scenario [Heintz et al., 2010b], images from color and thermal cameras are processed into a set of hypotheses about objects.

The object hypotheses need to be correlated over time to deal with the data association problem [Bar-Shalom and Fortmann, 1988]. There may be false positive and negative observations, temporal occlusions of objects and visually similar objects in the environment. One can reason also about the hypotheses based on, for instance, the normative characteristics of the physical objects they represent [Heintz et al., 2010a, Elfring et al., 2012]. For example, in the traffic monitoring scenario, one can consider the positions and speeds of objects perceived over time and the layout of the road network. This can be used to reason about stationary and moving objects and their types. For instance, when a car is observed again after a temporary occlusion, it should be assigned the same symbol before and after the occlusion.

The third situation pertains to flexible plan execution and monitoring in noisy and dynamic environments. The execution of actions/plans are to be driven, monitored and controlled by various conditions [Verma and Jónsson, 2006, Doherty et al., 2009, Ziafati et al., 2013a]. Conditions are monitored by low-level implementations of actions/behaviors to detect their success or failure. However, control and monitoring of plan execution via observation of various conditions at system-level is necessary.

The advantages of system level plan execution control and monitoring are to use data provided by different perception components to achieve system's goals, to avoid complicating implementation of actions and to avoid duplicating monitoring functionalities. Depending on an application, conditions to be monitored can be as simple as monitoring an object for being attached to the manipulator. They can be also complex logical, temporal and numerical conditions.

The fourth situation is high-level event recognition to recognize and react in real-time to situations in the environment. One example is detecting traffic violations such as reckless driving by observing qualitative spatial relations among cars [Heintz et al., 2013]. Another example is detecting situations and events such as "successful pass", "successful tackle" and "goal scoring" in football simulation or "washing hand before examination" and "basic clinical examinations carried out in time" in hospital simulations from lower

level events [Ranathunga et al., 2012]. The last example is recognizing human activities such as "cooking", "eating" and "watching TV" in smart homes [Pecora et al., 2012, Sabri et al., 2011]. Detecting such situations of the environment requires correlating and aggregating sensory data about changes of the environment based on their temporal and logical relations.

What all these situations have in common is a need for processing sensory data flows to extract new knowledge as soon as the relevant data becomes available without requiring persistent storage of data. Supporting on-flow processing requires an expressive and efficient language for real-time processing of data flows based on complex relations among the data items within the flows.

## 3.2 On-Flow Processing Systems

On-flow processing is an important requirement in various application domains [Cugola and Margara, 2012]. In environment monitoring, sensory data is processed to acquire information about the observed world, detect anomalies, or predict disasters. Financial applications analyze stock data to identify trends. Banking fraud detection and network intrusion detection require continuous processing of credit card transactions and network traffic, respectively. *RFID*-based inventory management requires continuous analysis of RFID readings. Manufacturing control systems often require observing system behavior to detect anomalies. As the result of many years of research from different research communities on such application domains, a large number of "information flow processing systems" have been developed to support on-flow processing of data [Cugola and Margara, 2012].

An extensive survey of information flow processing systems [Cugola and Margara, 2012] shows that the functionalities of these systems are converging to a set of operations and processing policies for on-flow filtering, combining and transformation of data, indicating universal usability of such functionalities for on-flow processing of data. This makes the existing information flow processing systems amenable to support on-flow information processing in robot software.

Disregarding the large amount of research existing on information flow processing systems and developing new on-flow processing systems for robotics from scratch will most probably end up with the development of similar systems to existing ones, adding to the tower of Babel syndrome for information flow processing systems [Etzion, 2007], negatively impacting the collaboration required to advance the state of the art [Cugola and Margara, 2012]. Robotic research should rather examine the usability of current

information flow systems for its different tasks to revise and extend these systems accordingly in order to satisfy its requirements and at the same time contribute to the field of information flow processing.

Current information flow processing research has led to two competing classes of systems [Cugola and Margara, 2012], Data Stream Management Systems (*DSMS*s) and Complex Event-Processing Systems" (*CEPS*s). *DSMS*s functionalities resemble database management systems. They process generic flow of data through a sequence of transformations based on common *SQL* operators like selections, aggregates and joins. Being an extension of database systems, *DSMS*s focus on producing query answers, which are continuously updated to adapt to the constantly changing contents of their input data. In contrast, *CEPS*s see flowing data items as notification of events happening in the external world. These events should be filtered and combined to detect occurrences of particular patterns of events representing higher level events. *CEPS*s are rooted in publish-subscribe model. They increase the expressive power of subscribing language in traditional publish-subscribe systems with the ability to specify complex event patterns.

Both *DSMS*s and *CEPS*s have their own merits and the recent proposals attempt to combine the best of both classes of systems [Cugola and Margara, 2012]. However, at this stage, the *CEPS*s are more suitable to support robotic on-flow processing due to the following reasons. First, the semantics given in *CEPS*s to data items as being event notifications naturally corresponds to time-stamped sensory data being observations of the environment by the robot perception components. Second, *CEPS*s put great emphasis on detection and notification of complex patterns of events involving sequence and ordering relations which constitutes a large number of robotic on-flow information engineering problems which is usually out of scope of *DSMS*s.

The rest of this chapter introduces *ETALIS*, a state-of-the-art *CEP*, and discusses its suitability for robotic on-flow information engineering through its comparison with related work.

## 3.3 *ETALIS* Language for Event-Processing (*ELE*)

The *ETALIS* language for event-processing (*ELE*)[1] [Anicic et al., 2012, 2010, Anicic, 2011] is an expressive and efficient language with formal declarative semantics for realizing complex event-processing functionalities. *ELE* advances the state-of-the-art CEP languages by allowing logical reasoning about domain knowledge in the specification of complex event patterns. Logical reasoning can be used to relate events, accomplish

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

complex filtering and classification of events and enrich events on the fly with relevant background knowledge.

### 3.3.1 ELE Syntax

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

**Definition 1 (*ELE* Signature [Anicic et al., 2012]).** A signature $\langle C, V, F_n, P_n^s, P_n^e \rangle$ 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 $P_n^s$ of static predicate symbols of arity $n$.
- For $n \in \mathbb{N}$ sets $P_n^e$ of event predicate symbols of arity $n$ with typical elements $p_n^e$, disjoint from $P_n^s$.

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

**Definition 2 (Term [Anicic et al., 2012]).** A term $t ::= c \mid v \mid f_n(t_1, ..., t_n) \mid p_n^s(t_1, ..., t_n)$.

**Definition 3 (Atom [Anicic et al., 2012]).** An static/event atom $a ::= p_n^{s/e}(t_1, ...t_n)$ where $p_n^{s/e}$ is a static/event predicate symbol and $t_1, ..., t_n$ are terms.

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

**Definition 4 (Event [Anicic et al., 2012]).** An event is a ground event atom time-stamped with an occurrence time.

- An atomic event refers to an instantaneous occurrence of interest.

- A complex event refers to an occurrence with duration.

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

**Definition 5 (*ELE* Rule [Anicic et al., 2012]).** An *ELE* rule is a static rule $r^s$ or an event rule $r^e$.

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

**Definition 6 (Event Pattern [Anicic et al., 2012]).** The language $P$ of event patterns is

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

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

### 3.3.2 ELE Semantics

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

**Definition 7 (Event Stream [Anicic et al., 2012]).** An event stream $\epsilon : Ground^e \rightarrow 2^{\mathbb{Q}^+}$ is a mapping from ground event atoms to sets of non-negative rational numbers.

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

**Definition 8 (ELE semantics [Anicic et al., 2012]).** Given an ELE program with a set $R$ of ELE rules, an event stream $\epsilon$, an event atom $a$ and two non-negative rational

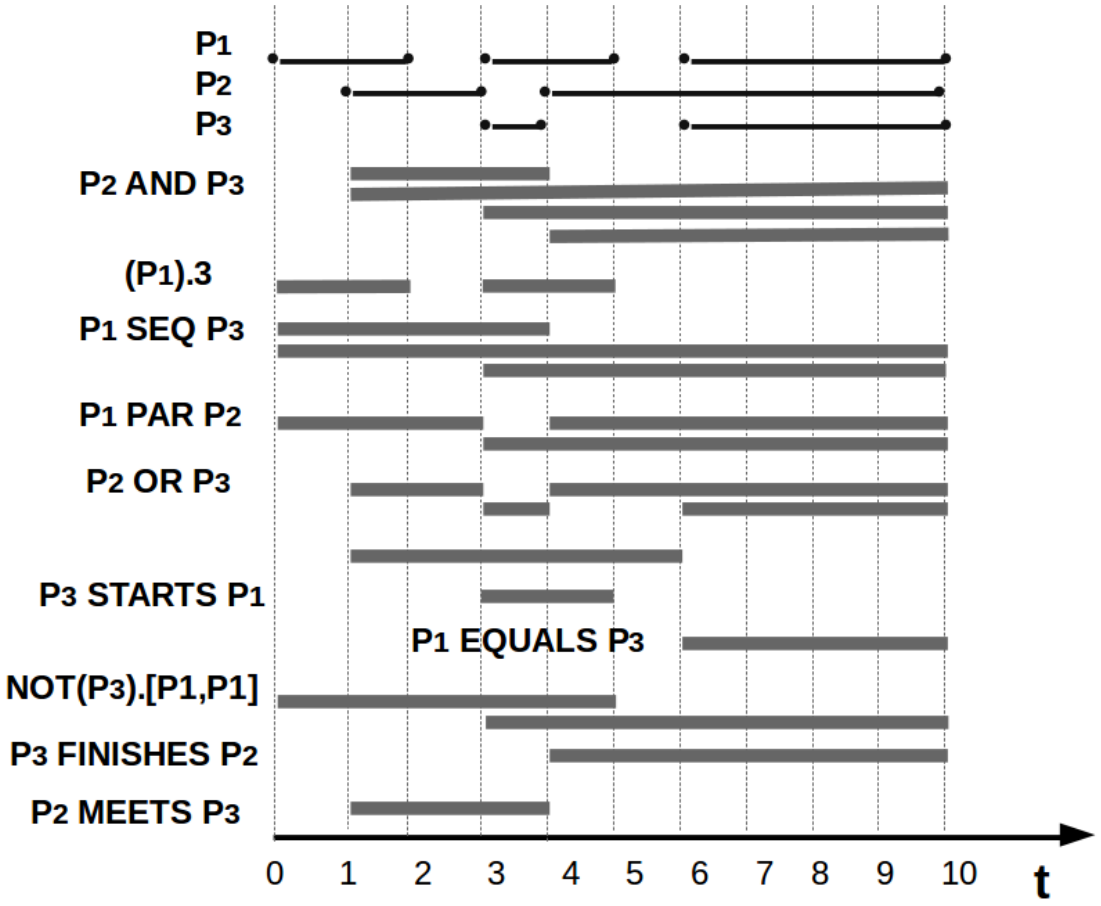


FIGURE 3.1: *ELE* event-processing operator examples, re-produced from [Anicic et al., 2012]

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 $\epsilon$ (i.e. $\epsilon, R \models a^{\langle q_1,q_2\rangle}$).

Figure 3.1 informally introduces the *ELE* semantics. It provides examples of how *ELE* operators are used to specify complex events in terms of simpler ones. The first three lines show occurrences of the instances of events $P_1$, $P_2$ and $P_3$ during time interval [0,10]. The vertical dashed lines represent units of system time and horizontal bars represent detected complex events for the given patterns. The presented patterns are read as follows:

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

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

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

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

This rule is read as follows. *S1* and *S2* are variables. When smoke is detected by a sensor *S1* and high temprature is detected by a sensor *S2*, a fire alarm event is generated, if these sensors are located nearby. If *P2* and *P3* in figure 3.1 represent smoke and high-temperature events from sensors located nearby, then a fire alarm is generated four times during the time interval [0,10].

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

Complex events are time stamped based on the temporal patterns they represent. For example, in Figure 3.1, the occurrence times of the first instances of *P2* and *P3* events are the intervals [1,3] and [3,4], respectively. According to *ELE* semantics, a fire alarm detected from these events is time stamped with time interval [1,4]. The time stamp of detected patterns can be used to filter the patterns. For example, a fire alarm should be

generated, only if both smoke and high temperature are detected within 300 seconds. This condition is added to the fire alarm pattern as follows.

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

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

*ELE* is free of operational side-effects, including the order between event-processing rules and delayed or out of order arrival of input events. For example, the sequence pattern in Figure 3.1 detects three events during the time interval [0,10], no matter the order in which *ELE* receives *P1* and *P3* events.

Listing 3.1 presents an *ELE* program to illustrate the modeling capabilities of the *ELE* language. In this program, the robot detects an event whenever a person moves an object. Such an event is detected when a person's face is observed while the object is moved.

The program is read piece by piece. The first clause generates a $see(f)$ event for every two immediate consecutive recognitions of a face $f$, occurring with confidence values over fifty within half a second. The variable $F$ is used to group the recognitions of faces in the event pattern and to pass information to generated events. The rule also explicitly encodes the start and end times of the sequence in content of the generated event by $T_s$ and $T_e$ variables.[2] The second clause detects reliable recognition of objects, when recognized three times within half a second with average confidence value over sixty. *pos_avg* is a static atom computing position of the object by averaging from its perceived positions. The third clause detects cases when an object is moved over five centimetres within a second. The fourth clause combines each two overlapping movement events of an object into a new one with a longer occurrence time. The fifth clause combines two

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

time periods of observing a person if they occur within three seconds after each other. Finally, the last clause detects when an object is moved during the time period a person is being observed.

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

Listing 3.1: An *ELE* program for monitoring objects moved by humans

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

The *ELE* language offers various *consumption policies* to filter our repetitive rule firings. These includes policies to select a particular pattern among possible matches and to limit the use of an event to fire a rule more than once. While such policies are not aligned with declarative semantics of *ELE*, they are widely adopted in *CEPS*s for practical reasons.

*ETALIS* also supports adding or deleting *ELE* rules at runtime allowing flexible reconfiguration of event-processing functionalities.

## 3.4 Runtime Subscription in *Retalis*

*ETALIS* interface facilitates programming a fixed set of output channels to deliver certain types of events from the events it processes, its input flow of events and the events it generates, to consumers. *Retalis* extends this functionality enabling robot software components to subscribe to *Retalis* for their events of interest at run-time. The events are sent to subscribers asynchronously as soon as they are processed by *Retalis*.

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

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

**Example.** When the robot is asked to follow the object segment $seg11$, the control component sets the target location for the Gaze component to the location of $seg11$

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

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

## 3.5 Performance

The *Etalis* execution model is based on decomposition of complex event patterns into intermediate binary event patterns (goals) and the compilation of goals into *goal-directed event-driven Prolog* rules. As relevant events occur, these rules are executed deriving corresponding goals progressing toward detecting complex event patterns. We will discuss the *Etalis* execution model in details in Chapter 6 and the performance of *Retalis* integrating the extension of *Etalis* with run-time subscription mechanism in Chapter 5.

Information flow processing systems such as *Etalis* are designed for applications that require a real-time processing of a large volume of data flow. We refer the reader to the detailed evaluation of the performance of *Etalis* presented elsewhere [Anicic et al., 2012, Anicic, 2011]. The evaluation shows, in terms of performance, *Etalis* is competitive with *ESPER*,[3] considered as a leading open source information flow processing system Cugola and Margara [2012]. The comparison is briefly outlined below.

On the basic event patterns $a\ SEQ\ b\ SEQ\ c$ and $NOT(c(ID, Z).[a(Id, X), b(Id, Y)]$ and $a(Id, X)\ AND\ b(Id, Y)\ AND\ c(Id, Z)$ and $a(Id, X)\ SEQ\ b(Id, Y)\ OR\ c(Id, Y)$, *Etalis*, executed by *YAProlog*[4] significantly outperforms *ESPER*. The performance is evaluated over input throughput, the number of input events processed per seconds, on a usual workstation. Input throughput for *ESPER* is reported to be between 10K to 20K and for *Etalis* is reported to be between 20k to 35k events per second. However, on the pattern $a(Id, X)\ SEQ\ b(Id, Y)\ WHERE(Y < K)$ where the parameter $K$ varies the selectivity of the $Y$ attribute, when the selectivity is in the range of 10 to 50%, *ESPER* significantly outperforms *Etalis*. This is due to *Etalis* evaluating the *WHERE* clause at the end. When the selectivity is 100%, *Etalis* performs slightly better than *ESPER*.

The performance of *Etalis* when detection of events includes reasoning tasks depends on the complexity of the corresponding *Prolog* queries and knowledge base. For instance, the execution of a pattern with throughput of 3900 events per second is reported in Anicic

[3] http://*ESPER*.codehaus.org/
[4] http://www.dcc.fc.up.pt/ vsc/Yap/

et al. [2012]. Each time the pattern processes an event, it accesses the RDF[5] data of 20k weather stations to find the latitude and longitude of the corresponding sensors, then computes the distance between two sensors and checks if the distance is less than a threshold. This shows that a large amount of background knowledge can be reasoned about in on-flow processing of events.

## 3.6 Related Work

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

In parallel to this research, the *DyKnow* [Heintz et al., 2010b, Heintz, 2009] framework has been extended with a number of tools that are relevant to on-flow processing [de Leng and Heintz, 2014, Heintz and Leng, 2013, Heintz, 2013]. The main feature of the work is the annotation of data streams and transformation processes with semantic descriptions. The semantic descriptions are used for automatic construction of streams of data. The *C-SPARQL* [Barbieri et al., 2010] language has been integrated to support the querying of flows of data. *C-SPARQL* belongs to the *DSMS*s category of on-flow processing systems. The advantages of *ETALIS* over *C-SPARQL* is its support for capturing complex data patterns. In contrast, *Retalis* does not support an automatic discovery of flows of data, for instance, required to detect a complex event. The input and output subscriptions of Information-Engineering Components and the event patterns they process are reconfigurable at runtime. However, such reconfigurations are not made automatic.

The literature does not contain a comparison between the expressive power of information flow processing systems. *ETALIS* is one of the most expressive systems as it supports most of the existing information flow processing operations listed in the survey of G. Cugola and A. Margara [Cugola and Margara, 2012]. In particular, *ETALIS* supports the representation of all possible thirteen temporal relations between time interval occurrence times of two events as defined in Allen's interval algebra [Allen, 1983], non-occurrence of an event between the occurrence of two other events, and iterative and aggregating patterns. Furthermore, arbitrary processes can be applied on events through the use of static atoms in *ETALIS* syntax, provided that such processes are interfaced

[5] http://www.w3.org/RDF/

with the *Prolog* language. An example is interfacing spatial reasoning functionalities with *Prolog* presented by M. Tenorth and M. Beets [Tenorth and Beetz, 2012].

Logic-based approaches such as *Chronicle Recognition* [Ghallab, 1996] and *Event Calculus* [Kowalski and Sergot, 1989, Shanahan, 1999] have received considerable attention for event representation and recognition due to their merits, including expressiveness, formal and declarative semantics and being supported by machine learning tools to automate the construction and refinement of event recognition rules [Artikis et al., 2010, Anicic et al., 2012]. However, the query-response execution mode and scalability of classic logic-based systems limits their usability for on-flow information processing. The query-response execution means detecting an event at runtime requires frequently querying the system for that event. Moreover, the event is detected only when the next time the system is queried for that event. In addition, efficient evaluation of such queries requires caching mechanisms not to re-evaluate queries over all historic data [Chittaro and Montanari, 1996]. *ETALIS* bridges the gap between *CEPS*s and logic-based event-recognition systems by offering a logic-based *CEPS* with an event-driven, incremental and efficient execution model.

The *IDA* [Wrede, 2009, Lütkebohle et al.] and *CAST* [Hawes and Wyatt, 2010, Hawes et al., 2008] are robotic frameworks supporting the subscription of components to their events of interest based on the type and content of events. Using *XML* data format, a subscriber can register for information items containing specific field of data. *IDA* also provides few types of event filters such as the *Frequency filter*, which outputs only every $n$-th received notification. *Retalis* provides a general framework to address a much wider variety of event processing requirements, including temporal and spatial reasoning over events to detect complex event patterns. Moreover, the subscription mechanisms of *IDA* and *CAST* are tightly built over their underlying middleware. In contrast, *Retalis* is framework-independent and has been interfaced with *ROS* which is widely used by robotic community.

The use of *CEPS*s for detecting high-level events in agent research has been proposed before. Buford et al. [Buford et al., 2006] extend the *BDI* architecture with situation management components for event correlation in distributed large-scale systems. Ranathunga et al. [Ranathunga et al., 2012] utilize the *ESPER*[6] event-processing language to detect high-level events in second life virtual environments.[7] However this work is not concerned with the robotic on-flow information-processing problem, it does not provide a formal account of event processing and does not support run-time subscription. Other related work includes various approaches for high-level event recognition,

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

[7]http://secondlife.com

anchoring and monitoring, for instance, using Chronicle recognition, constraint satisfaction or variants of temporal logic [Heintz et al., 2010b, Pecora et al., 2012, Heintz et al., 2013, Doherty et al., 2014]. Such approaches do not satisfy all on-flow information processing requirements. For instance, the Chronicle recognition or constrained satisfaction approaches based on simple temporal networks cannot express atemporal constraints, and temporal logic based approaches do not support transformation of information.

## 3.7 Summary

From an analysis of four robotic situations including component interactions, plan execution and monitoring, anchoring and situation recognition, we generalize the on-flow processing requirements as follows. On-flow processing requires on the fly processing of sensory data flows to extract knowledge as soon as the relevant data becomes available without requiring, at least in principle, persistent storage of data. Supporting on-flow processing requires an expressive and efficient language for real-time processing of data flows based on complex temporal and logical relations among the data within the flows. Addressing these requirements is the focus of information flow processing systems applied in various domains such as banking fraud and network intrusion detection and stock data analysis. Among these systems, we integrate the *Etalis* complex event-processing system to support on-flow processing in *Retalis*.

The *ELE* language of *Etalis* is an expressive and efficient language with formal declarative semantics for realizing complex event-processing functionalities. *ELE* advances the state-of-the-art information flow processing systems by allowing logical reasoning about domain knowledge in the specification of complex event patterns. *ELE* is one of the most expressive on-flow information processing systems as it supports the representation of all possible thirteen temporal relations among time interval occurrence times of two events, non-occurrence of an events between the occurrence of two other events, and iterative and aggregating patterns. Moreover, parsed and executed by *Prolog*, interfaces of the underlying *Prolog* execution system with other languages can be used to interface *ELE* with other systems and libraries, for instance, to perform spatial reasoning on patterns of events. Despite its expressiveness and its execution by *Prolog*, it has been shown that *Etalis* achieves a performance of competitive with state-of-the-art information-flow processing systems such as *ESPER*.

We extend *Etalis* with a run-time subscription mechanism and interface it with *ROS*. This allows *ROS* components to subscribe to *Etalis* for their events of interest at run-time, filtered by logical conditions on the contents of events. It allows also to subscribe *Etalis* to messages from *ROS* components. The conversion between *ROS* messages and

*Etalis* events are performed automatically. Supporting runtime subscription, flows of data among components can be configured according to runtime operational contexts of the robot to change the robot's behavior or save resources by communicating only the relevant data. By integrating *Etalis* and extending it with a runtime subscription mechanism, *Retalis* provides an advanced support for on-flow processing in robotics, considerably improving over the existing systems, shown by providing its detailed comparison with existing systems and approaches.

# Chapter 4

# On-Demand Information Processing

The question of this chapter is how to support on-demand processing of information. An extensive study of robotics knowledge management requirements highlights the dominant advantages of logic-based systems [Lemaignan, 2012]. The question is how to address the limitation of existing systems with regard to selecting, managing, querying and synchronizing the relevant parts of flows of sensory data in the knowledge base and support active memory notifications. Language support is required to enable the definition of high-level policies for selective recording of data and pruning outdated data. It is also required to facilitate the state-based representation of data built upon discrete observations of the environment (i.e. events). An efficient management and querying of histories of data requires their underlying management in suitable data structures and using indexing mechanisms. Active memory notifications requires support for generation and management of events. Finally, language support is needed to synchronize queries on the state of the environment, built upon events, asynchronously received from the perception components.

This chapter is organized as follows. We first briefly discuss the on-demand processing requirements related to representation, management, querying and synchronization of the discrete and asynchronous flows of sensory information continuously generated by the robot's perception components in the knowledge base. We then present the *SLR* language [Ziafati et al., 2014], developed in this thesis, to address these on-demand processing requirements that are not satisfactorily supported by existing systems. The *SLR* syntax and semantics are presented and the usability of the language and its relation with existing work is discussed. Finally, a summary is given.

## 4.1 On-Demand Processing Requirements

On-demand information processing corresponds to managing data in memory or knowledge base to be queried and reasoned upon on request. A large set of on-demand information processing requirements has been discussed elsewhere [Lemaignan, 2012, Wrede, 2009]. Consequently, in this section we only discuss the on-demand processing requirements related to discreteness, asynchronicity and continuity of robotic sensory data that are not satisfactorily supported by existing systems.

Robot knowledge includes knowledge of its domain, common-sense knowledge and knowledge of the dynamics of its world collected by its perception components. Perception components 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 et al., 2010b, Wrede, 2009]. The robot knowledge collected by events, representing observations of the environment and time-stamped with the time of their occurrence, needs to be properly represented and maintained to reason on. However, the discrete and asynchronous nature of observations and the continuous generation of events make querying and reasoning on such knowledge difficult and pose many challenges on their use, for instance, in robot task execution.

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

Building robot knowledge of its environment upon sensory events requires language support to simplify reasoning about the state of the environment at a time based on discrete observations of the environment.

A network of distributed and parallel components processes robot sensory data and sends the resulting events to the knowledge base. Due to processing times of the perception components and possible network delays, the knowledge base may receive the events with some delays and not necessarily in the order of their occurrence. For example, the event indicating the recognition of an object in a *3D* image is generated by the object recognition component sometime after the actual time at which the object is

observed, because of object recognition processing time. Another example is when data is generated or needs to be verified by an external source with arbitrary operating time.

Dealing with asynchronicity of sensory data requires supporting the implementation of synchronization mechanisms to assure evaluating queries when relevant data to queries are available in the knowledge base. When the knowledge base is queried, correct evaluation of the query may require waiting for the perception components to finish processing of sensory data to ensure that all data necessary to evaluate the query is present in the knowledge base. For example, the query, "how many cups are on the table at time $t$?" should not be answered immediately at time $t$, but answering the query should be delayed until after completing the processing of pictures of the table by the object recognition component and the reception of the results by the knowledge base.

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

## 4.2 *SLR* Language for Event Management and Querying

Synchronized Logical Reasoning language (*SLR*) [Ziafati et al., 2014] is a knowledge management and querying language for robot software enabling the high-level representation, querying and maintenance of robot knowledge. In particular, *SLR* aims at simplifying the representation of robot knowledge based on its discrete and asynchronous observations and improving efficiency and accuracy of query evaluation by providing synchronization and event-history management mechanisms. These mechanisms facilitate ensuring that all data necessary to answer a query is gathered before the query is answered and that outdated and unnecessary data is removed from memory.

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

In *Prolog* syntax, a *term* is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a functor symbol and $t_1, \ldots, t_n$ are constants, variables or terms. A term is *ground* if it contains no variables. A *Horn clause* is of the form $a_1 \wedge \ldots \wedge a_n \rightarrow a$, where $a$ is a term

called the *Head* of the clause, and $a_1, \ldots, a_n$ is called the *Body* where $a_i$ are terms. In *Prolog* syntax, the body can also include negation of terms. $a \leftarrow true$ is called a *fact* and usually written as $a$. A *Prolog program* $P$ is a finite set of *Horn clauses*.

One executes a logic program by asking it a query. *Prolog* employs the *SLDNF* resolution method [Apt and van Emden, 1982] to determine whether or not a query follows from the program. Given a goal, *SLDNF* tries to prove the goal using the rules and facts of the program. A goal is proved if there is a variable substitution by applying which the goal matches a fact, or matches the head of a rule and the goals in body of the rule can be proved from left to right. If a goal is a negation of a term, it succeeds when the term can not be proved (i.e. negation as failure). Goals are resolved by trying the facts and rules in the order they appear in the program. A query may result in a substitution of free variables. We use $P \vdash_{SLDNF} Q\theta$ to denote a query $Q$ on a program $P$, resulting in a substitution $\theta$.

### 4.2.1 *SLR* Syntax

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

**Definition 9 (*SLR* Signature).** A signature $S = \langle C, R, V, Z, P^s, P^e \rangle$ for *SLR* language consists of:

- A set $C$ of *constant symbols*.
- A set $R \subseteq \mathbb{R}$ of *real numbers*.
- A set $V$ of *variables*.
- A set $Z \subseteq R_{r \geq 0} \cup V$ of *time points*
- $P^s$, a set of $P^s_n$ of *static functor symbols* of arity $n$ for $n \in \mathbb{N}$.
- $P^e$, a set of $P^e_n$ of *event functor symbols* of arity $n$ for $n \in \mathbb{N}_{n \geq 2}$, disjoint with $P^s_n$.

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

For the sake of readability, an event term is denoted as $p_n(t_1, \ldots, t_{n-2})^{[z_1,z_2]}$. Moreover, an event term whose $z_1$ and $z_2$ are identical is denoted as $p_n(t_1, \ldots, t_{n-2})^{z}$ ($Z = Z_1 = Z_2$).

**Definition 11 (Event).** An *event* is a ground event term $p_n(t_1, ...t_n)^{[z_1,z_2]}$, where $z_1$ is called the *start time* of the event and $z_2$ is called its *end time*. The functor symbol $p_n$ of an event is called its *event type*.[1]

We introduce two types of static terms, *next* and *prev* which respectively refer to occurrence of an event of a certain type observed right after and right before a time point, if such an event exists. In the next section we provide the semantics. In this section, we restrict ourselves to the syntax of *SLR*.

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

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

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

### 4.2.2 *SLR* Semantics

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

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

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.

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

**Definition 16 (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 time point $z$, i.e. $\epsilon(z) = \{p_n(t_1, \ldots, t_{n-2})^{[z_1,z_2]} \in \epsilon \mid z_2 \leq z\}$.

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

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

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

The $z$ parameter of a query sets the scope of the query to set of observations made up until time $z$. This means that the query $\langle Q, z \rangle$ cannot be evaluated before time $z$, since *SLR* would not have received the robot's observations necessary to evaluate $Q$ and the query can be evaluated as soon as all observations up to time $z$ is in place. The only exceptions are the *prev* and *next* clauses whose evaluation might need observations made after time $z$.

A query $\langle Q, z \rangle$ can be posted to *SLR* long after time $z$, in which case the *SLR* knowledge base contains observations made after time $z$. In order to have a clear semantics of queries, *SLR* evaluates a query $\langle Q, z \rangle$ by only taking into account the event facts in $\epsilon(z)$. Regardless of the $z$ parameters of queries, the *next* or *prev* clauses are evaluated based on their declarative definitions as follows.

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

*prev* clause succeeds and fails otherwise.

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

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

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

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

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

The *next* and *prev* clauses can be implemented by the following two *Prolog* rules. However, we take advantage of the fact that *SLR* usually receives events of the same type in the order of their end times. *SLR* maintains the sorted list of events of each type ordered by their end times whose maintenance usually only requiring the assertion of events by

the *asserta Prolog* built-in predicate. In this way, finding a previous/next event of a type occurring before/after a time point requires examining only a part of the history of those events. The ¬ symbol represents *Negation as failure*.

$$
\begin{aligned}
&\mathit{prev}(p_n(t_1,...t_n)^{[z_1,z_2]}, z_s)\text{:-}\\
&\qquad p_n(t_1,...t_n)^{[z_1,z_2]},\\
&\qquad z_2 \leq z_s,\\
&\qquad \neg(p_n(t_1",...t_n")^{[z_1",z_2"]},\ z_2" \leq z_s, z_2" > z_2). \qquad (4.1)\\
&\mathit{next}(p_n(t_1,...t_n)^{[z_1,z_2]}, z_s, z_e)\text{:-}\\
&\qquad p_n(t_1,...t_n)^{[z_1,z_2]},\\
&\qquad z_s \leq z_2 \leq z_e,\\
&\qquad \neg(p_n(t_1",...t_n")^{[z_1",z_2"]},\ z_s \leq z_2" \leq z_e, z_2" < z_2). \qquad (4.2)
\end{aligned}
$$

### 4.2.3 State-Based Knowledge Representation

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

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

**Example.** 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.

$$\mathit{color}(O,C)^T\text{:-}\ \mathit{prev}(\mathit{obj}(O,,C)^Z, T). \qquad (4.3)$$

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

**Example.** To pick up an object $O$, its location should be determined and sent to a planner to produce a trajectory for the manipulator to perform the action. This task can be naively presented as the sequence of actions: determine the object's location $L$, compute a manipulation trajectory *Trj*, and perform the manipulation. However, due to environment dynamics and interleaving in task execution, the robot needs to check that the object's location has not been changed and the computed trajectory is still valid before executing the actual manipulation task. The following three rules can be used to

determine the location of an object and its validity as follows. If the last observation of the object is within the last five seconds, the object location is set to the location at which the object was seen last time. If the last observation was made longer than five seconds ago, the second rule specifies that the location is outdated. The third rule sets the location to "never-observed", if the robot has never observed such an object. The symbol ! represents *Prolog* cut operator and locations are assumed to be absolute.

$$\begin{aligned} &\mathit{location}(O, L)^T\text{:-} \\ &\qquad \mathit{prev}(\mathit{seg}(O, L)^Z, T), T - Z \leq 5, !. \qquad (4.4) \\ &\mathit{location}(O, \text{“}\mathit{outdated}\text{”})^T\text{:-} \\ &\qquad \mathit{prev}(\mathit{seg}(O, L)^Z, T),\ T - Z > 5, !. \qquad (4.5) \\ &\mathit{location}(O, \text{“}\mathit{never\text{-}observed}\text{”})^T. \qquad (4.6) \end{aligned}$$

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

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

$$\begin{aligned} &\mathit{tf}(\mathit{cam}, \mathit{base}, L)^T\text{:-} \\ &\qquad \mathit{prev}(\mathit{tf}(\mathit{cam}, \mathit{base}, L_1)^{T_1}, T), \\ &\qquad \mathit{next}(\mathit{tf}(\mathit{cam}, \mathit{base}, L_2)^{T_2}, [T, \infty]), \\ &\qquad \mathit{est}([L, T], [L_1, T_1], [L_2, T_2]). \qquad (4.7) \end{aligned}$$

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

$$
\begin{aligned}
&tf(base, rcf, L)^T \text{ :-} \\
&\qquad prev(tf(base, rcf, L_1)^{T_1}, T), \\
&\qquad ( \\
&\qquad next(tf(base, rcf, L_2)^{T_2}, T, T+1) \rightarrow \\
&\qquad\qquad est([L, T], [L_1, L_2], [L_2, T_2]) \\
&\qquad\qquad ; \\
&\qquad\qquad L \; is \; L_1 \\
&\qquad ).
\end{aligned} \tag{4.8}
$$

The following *ELE* rule concerns recognition of an object $O$ at a position $L_{o-c}$ relative to the camera at a time $T$. It generates a corresponding *segR* event. It calculates the object position in the reference coordination frame by querying the *SLR* knowledge base. The camera to base and base to world relative positions at time $T$ are estimated by rules (13) and (14).

$$
\begin{aligned}
&segR(O, L) \leftarrow \\
&\qquad seg(O, L_{o-c})^T \\
&\qquad WHERE( \\
&\qquad\qquad tf(cam, base, L_{c-b})^T, \\
&\qquad\qquad tf(base, rcf, L_{b-rcf})^T, \\
&\qquad\qquad mul([L_{o-c}, L_{c-b}, L_{b-rcf}], L) \\
&\qquad\qquad ).
\end{aligned} \tag{4.9}
$$

### 4.2.4 Active Memory

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

**Definition 21 (Memory Instance).** A memory instance with an id $Id$, a query $Q$ and a policy $\langle L, N \rangle$ keeps the record of a subset of input events to *SLR*: the events that match the query $Q$ such that at each time $T$, the memory instance only contains the events which have their end times within the last $L$ seconds and only includes the recent $N$ number of such events ordered by their end time. An id is a ground term and a query is of the form $\langle e, Cond \rangle$, where $e$ is an event atom and $Cond$ is a set of conditions on

variables that are arguments of $e$. An event $P$ matches a query pattern $Q$ when there is a substitution that can unify $p$ and $e$ and makes the conditions in *Cond* true (i.e. $\exists\theta(p = q_\theta)$).

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

**Example.** The $c\_mem(tf, \langle tf(X,Y,Z), \langle\rangle\rangle, \infty, 300)$ query creates a memory instance to keep the history of *tf(X,Y,Z)*$^T$ events from the *stateRec* component for 300 seconds. In the rule (15), we saw that the *SLR* knowledge base is queried to position object segments in the reference coordination frame. If we assume that the *IEC* receives data of object segments within 300 seconds since they appear in front of the camera, then we only need to keep the history of *tf* events for 300 seconds. In another example, for each object $o_i$ in *segR(O,L)* events, the *ELE* rule (16) generates a memory instance with the corresponding id of $obj(o_i)$. A memory instance is generated, if it does not already exist. This is checked using the $\neg exist\_mem(monitor(O))$ clause. Each memory instance $obj(o_i)$ keeps the last occurrence of $segR(o_i, L)$ events at which $o_i$ is located on the floor, checked by the *onFloor Prolog* term implementing the required spatial inference. The use of *DO* clause is another way of performing *SLR* queries in *Etalis* syntax.

$$\begin{aligned} Do(c\_mem(obj(O), \langle segR(X,L), \langle X == O, onFloor(L)\rangle\rangle, 1, \infty)) \\ \leftarrow segR(O,L) \\ WHERE(\neg exist\_mem(obj(O))). \end{aligned} \tag{4.10}$$

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

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

**Definition 23 (Previous_Memory Term Semantics).** A $prev(Id, X, Z_s)$ term, where $Id$ is a ground term, unifies $X$ with an event which has the latest occurrence time among

the events that belong to the memory instance $Id$, are unifiable with $X$ and have their end time before or equal to $Z_s$. The term fails if such a unification is not found.

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

**Example.** The rule (4.11) re-writes the rule (4.8) by querying the previous *tf(base,rcf,L)* event occurring before $T$ and the next *tf(base,rcf,L)* event occurring during $[T, T+1]$ from the memory instance *tf*, defined in the previous example to keep the history of *tf* events for 300 seconds. Another example is the query *findAll*(X, *mem(obj(O),X)*, *List*) which queries all $obj(O)$ memory instances created by the rule (4.10) for their records of $segR(X, L)$ events using the $mem(obj(O), X)$ template and put the list of results in the variable *List*.

$$
\begin{aligned}
&tf(base, rcf, L)^T \text{ :-} \\
&\qquad prev(tf, tf(base, rcf, L_1)^{T_1}, T), \\
&\qquad ( \\
&\qquad\ next(tf, tf(base, rcf, L_2)^{T_2}, T, T+1) \rightarrow \\
&\qquad\qquad\qquad est([L, T], [L_1, L_2], [L_2, T_2]) \\
&\qquad\qquad\qquad ; \\
&\qquad\qquad\qquad L \ is \ L_1 \\
&\qquad ).
\end{aligned}
\tag{4.11}
$$

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

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

### 4.2.5 Synchronizing Queries over Asynchronous Events

*SLR* supports the synchronization of queries to deal with the delayed and out of order reception of sensory data to the knowledge base. A distributed and parallel network of components with varying operating times processes robot sensory data. Therefore, an event, containing information extracted by these components about an observation of the environment, is generated at some time after the event occurrence time. Consider a picture taken at a time $t$. If we assume it takes $x$ second for the *segRec* component to process the image, then the corresponding $seg(o_i, c_j, p_k, l_g, pcl_h)$ events (i.e. recognized object segments) are generated at time $t + x$ but are time-stamped with time $t$. The *IEC* receives these events over a network and sends those object segments that their types have not yet been recognized with a high certainty to the *objRec* component. The *objRec* component, which can be, for instance, a cloud web service [Kehoe et al., 2013] receives such an event at some later time $t+x+y$. Then, the object segment is processed for its type and the corresponding $obj(o_i, ot_j, p_k)$ event is generated and sent to the *IEC*. The event is time-stamped with time $t$ but received by the *IEC* at some later time $t + x + y + z$.

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

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

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

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

Moreover, all previous and next memory terms should be correctly evaluated according to their definitions. Finding the previous event of type $p_n(t_1, .., t_n)$ occurring up to

time $z_s$ requires having received all $p_n(t_1, .., t_n)$ events occurring up until time $z_s$. If we assume events of each type are received by *SLR* in the order of their end times, then finding the next event of type $p_n(t_1, .., t_n)$ occurring within time interval $[z_s, z_e]$ requires having received the first $p_n(t_1, .., t_n)$ event which has its end time equal or more than $z_s$, or make sure that no $p_n(t_1, .., t_n)$ event has occurred during $[z_s, z_e]$. *SLR* postpones an individual query[2] when necessary until it is achievable, as defined below.

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

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

To determine when the history of events of a type $p_n$ up to a time $z$ is available, *SLR* can be programmed in two complementary ways.

The first way is to set a maximum delay time (i.e. $t_{d_{max}}$) for events of each type. When the system time passes $t_{d_{max}}(p_n)$ seconds after $z$, *SLR* assumes that the history of events of type $p_n$ up to time $z$ is available. The maximum delay times of events depends on the runtime of the components generating them and need to be approximated. The maximum delay times can be set the system developer. It can also be approximated by *SLR* as follows. Whenever an event of type $p_n$ is processed, *SLR* checks its delay, the difference between its end time and the current system time, and sets the $t_{d_{max}}(p_n)$ to the maximum delay time of $p_n$ events encountered so far.

When smaller maximum delay times of events are assumed, queries are evaluated sooner and hence the overall system works in more real-time fashion, but there is more chance of answering a query when the complete history of events asked by the query is not in place yet. When larger maximum delay times of events are assumed, there is a higher

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

chance to have all sensory data up to the time specified by the query already processed by the corresponding components and their results received by *SLR* when the query is evaluated. However, queries are performed with more delays.

The second way of programming *SLR* to determine the availability of an event history is as follows. *SLR* can ensure to have received the full history of events of a type $p_n$ up to a time $z$, when it is told so by a component generating such events using special $updated(p_n)^z$ events. Whenever *SLR* receives such an event, it assumes that the history of events of the type $p_n$ up to time $z$ is available and proceeds with executing the relevant queries.

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

**Example.** When the position of an object $O$ in the world coordination frame at a time $T$ is queried by the rule (15), the query can be answered as soon as both camera to base and base to world relative positions at time $T$ can be evaluated by rules (13) and (14). The former can be evaluated (i.e. interpolated) as soon as *SLR* receives the first $tf(\text{`}cam\text{'}, \text{`}base\text{'}, P)$ event with a start time equal or later than $T$. The latter can be evaluated as soon as the *SLR* receives the first $tf(\text{`}base\text{'}, \text{`}world\text{'}, P)$ event with the start time equal or later than $T$, or when it can ensure that no $tf(\text{`}base\text{'}, \text{`}world\text{'}, P)$ event has occurred within $[T, T+1]$. If we assume $t_{d_{max}}(tf(\text{`}base\text{'}, \text{`}world\text{'}, P))$ is set to 0.5 second, *SLR* has to wait 1.5 second after $T$ to ensure this.

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

$$
\begin{aligned}
&findall( \\
&\quad obj(O, Type, L), \\
&\quad (\ mem(obj(O), segR(O, L)^{T_x}),\ t_1 \leq T_x \leq t_2,\ prev(obj(O, Type, P)^{T_y}, t_2)\ ), \\
&\quad List \\
&\quad ) \qquad (4.12)
\end{aligned}
$$

The query result is the list *List* of terms of the form *obj(O,Type,L)* matching the template specified by the second argument of the *findall* term. This includes all object segments

recorded as *segR(O,L)*$^{T_x}$ events in *obj*$(O)$ memory instances recognized during $[t_1, t_2]$. The type of each object segment $o_i$ is recognized by querying the last *obj($o_i$,Type,P)* event occurring before or at time $t_2$.

To list all the objects, *SLR* makes sure to evaluate the query after the histories of both *segR(O,L)* and *obj(O,Type,L)* events up to time $t_2$ are available. A signaling mechanism to realize this is as follows. After finishing the processing of each image taken at a time $t$ and outputting the recognized object segments, the *segRec* component sends out the event *updated(segR)*$^t$. The *IEC* receives these events sending object segments whose type is not known and the *updated(segR)*$^t$ events to the *objRec* component. We assume events of each type are communicated among the components in order. The *objRec* component receives some object segments recognized at a time $t$, processes them in the order it receives them and sends the recognized types back to the *IEC*. Whenever the *objRec* processes an *updated(segR)*$^t$ event, it realizes that it has finished processing of the object segments recognized up to time $t$ and generates an *updated(obj)*$^t$ event. Receiving *updated(segR)*$^t$ and *updated(obj)*$^t$ events, *SLR* is notified when the histories of both types of events up to time $t_2$ are available and then evaluates the query.

## 4.3 Related Work

The use of memory in existing research includes collecting data from various sources and in time, mediating as a shared resource for component interaction (i.e. blackboard architectural pattern [Watson, 1990]), refining data by various processes, and integrating various reasoning capabilities to maintain and query the robot's knowledge of the environment for task execution, human interaction and learning [Bauckhage et al., 2008, Wrede, 2009, S. Wrede, M. Hanheide et al., 2004, Hawes and Wyatt, 2010, Hawes et al., 2008, Tenorth and Beetz, 2009, 2012, Lemaignan et al., 2011, Lemaignan, 2012, Lim et al., 2011, Mavridis and Deb Roy, 2006]. A large set of on-demand information processing requirements have been discussed elsewhere [Lemaignan, 2012, Wrede, 2009].

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

The advantage of non logic-based data representations, for example, using programming data structures in *CAST* [Hawes et al., 2008, Tenorth and Beetz, 2009] and

*GSM* [Mavridis and Deb Roy, 2006], is the flexibility and efficiency in the representation and manipulation of amodal data such as image data and probability distributions. However the expressiveness of queries for information maintained by such systems is limited. An interesting approach is the XML data representation by *IDA* [Wrede, 2009, S. Wrede, M. Hanheide et al., 2004] supporting *Xpath* queries [Birbeck, 2001], for example, to retrieve data of objects recognized with confidence of more than a threshold. The data representation in non-logic based systems is usually tightly related to the data representation used in their underlying framework and does not support logical reasoning. In *Retalis*, binary data is represented as *String*. This requires encoding binary data to the *Prolog String* format when importing a *ROS* message to *Retalis* and decoding it when the data is sent back to *ROS*, which is time consuming. However, one can maintain the actual binary objects in *c++* and manipulate handlers to the objects in *Retalis*.

A recent survey of existing robotic *information management* systems [Lemaignan, 2012] shows that most systems rely on logical formalisms, mainly including declarative languages such as the *OWL*[3] language [Mike Dean and Stein, 2004] based on Description logics [Baader et al., 2008] and/or rule-based languages such as the *SWRL*[4] language [Horrocks et al., 2004] for rule-based reasoning in *OWL* and *Prolog*. In particular, *OWL* is a popular choice to define ontologies of various types of knowledge such as knowledge of space, objects, actions and robot capabilities used, for instance, in *ORO* [Lemaignan et al., 2011, Lemaignan, 2012], *KnowRob* [Tenorth and Beetz, 2012, 2009] and *OUR-K* [Lim et al., 2011].

Defining ontologies are necessary to integrate various sources of knowledge such as the domain and common sense knowledge as performed by the aforementioned systems and for sharing robots' knowledge, for instance, in the cloud [Tenorth et al., 2012]. While we did not address modeling of knowledge, existing ontologies can be directly used in *Retalis* as *OWL* ontologies can be represented and reasoned upon in *Prolog*. For example, *KnowRob* offers one of the most comprehensive robotic ontologies and uses the *Prolog Semantic Web Library*[5] [Polleres et al., 2007] for loading and storing *RDF*[6] [Candan et al., 2001] triples and the *Thea*[7] *OWL* parser library [Vassiliadis et al., 2009] for OWL reasoning on top of this representation.

The use of Prolog as the underlying technology for maintaining robotic *OWL* knowledge has a few practical advantages for inference compared to the use of existing description logic reasoners such as the *Pellet*[8] reasoner [Sirin et al., 2007] used in *ORO*. Those

[3] http://www.w3.org/TR/2004/REC-owl-ref-20040210/
[4] http://www.w3.org/Submission/SWRL/
[5] http://www.swi-prolog.org/pldoc/package/semweb.html
[6] http://www.w3.org/RDF/
[7] http://www.semanticweb.gr/thea/
[8] http://clarkparsia.com/pellet/

reasoners keep a classified version of the knowledge base in memory specifying each individual belonging to which classes. Therefore continuous changes of the knowledge base through acquiring sensory data requires frequent re-classification of the whole knowledge which can be costly [Tenorth and Beetz, 2012]. This problem can be partially addressed by optimizing this operation using an incremental updating technique [Halashek-Wiener et al., 2006].

The more important advantage is related to the open world assumption in *Description Logics* versus the closed world assumption in Prolog, and the monotonicity of description logics versus supporting a form of non-monotonicity in *Prolog* by the *negation as failure* inference rule within the closed world assumption. In the closed world assumption, representations can be more compact as 'a fact not being true' does not need to be described but it can be inferred by not being able to prove the fact. Moreover, the open world assumption and monotonicity of *Description Logic* makes the representation and reasoning on dynamics of the environment (i.e. changes and actions) difficult requiring to handle such aspects externally [Ziafati et al., 2011, Lemaignan, 2012], but, for instance, *KnowRob* implements a predicate to return an object's location at a time by searching for the last observation of the object's location before that time.

Reasoning about changes and actions has been extensively studied in various knowledge formalisms such as Situation Calculus [Levesque et al., 1998] and *Event Calculus* [Kowalski and Sergot, 1989, Shanahan, 1999]. The *SLR* language provides a practical and efficient solution for representing robot knowledge based on discrete observations, providing a means to deal with the temporal validity of data and representation of continuous domains which is not the focus of such formalisms. Compared to the *KnowRob* approach of, for instance, implementing a predicate to represent an object's location at a time, *SLR* simplifies the definition of such predicates in general and increases the efficiency of their computations by maintaining the sorted list of events based on their occurrence times.

*Prolog* provides a flexible support for access to external data or reasoning functionalities while reasoning on knowledge through procedural attachments to the *Prolog* terms. This feature is used in *KnowRob*, for instance, to compute spatial relations between objects and in *Retalis* to integrate *OpenGL Mathematics*[9] (GLM) for arithmetic operations.

The *SLR* support for synchronization of queries on knowledge built upon asynchronous data is not presented elsewhere. However, similar synchronization mechanisms as found in *SLR* are implemented in other robotic software in a more limited context. One example is the *DyKnow* framework [Heintz, 2009] that synchronizes data received from streams of data based on different policies to generate new ones. Another example is

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

the *tf* library [Foote, 2013] widely used in *ROS* for querying position transformation between robot's coordination frames over time. When a relative position at a time is queried, the query is not answered until receiving the first observation of that position at or after that time. The *tf* library only supports interpolation of data similar to the *SLR* rule (13). Therefore, even if a position is constant in time, its value needs to be continuously published to *ROS* consuming the network bandwidth. Moreover, sometimes a component such as *AMCL* in *ROS* provides updates in a slow rate but they are precise enough to be used until the next update is made available. In order to not delay the processing of data until availability of the next update, this component stamps its updates in the future.[10] Apart from being semantically confusing, time stamping updates in future can result in using old data even if new data is already available. With the *SLR* extrapolation approach, for instance, implemented by the rule (14), if a position transformation is static, its value does not need to be published being extrapolated from its last observed value. In addition, the time bound of the *next* predicate in *SLR* allows to specify how long *SLR* needs to wait to see whether a value has been changed, assuming after each relevant change a notification is received.

Except a few, most information management systems leave pruning data from the memory to external components. In *ORO*, knowledge is stored in different memory profiles, each keeping data for a certain period of time. In *IDA*, scripts are activated periodically or in response to events of memory changes to perform garbage collection. In *SLR*, flexible garbage collection functionalities are blended in the syntax of the language. In addition, a subtle difference between *SLR* and other systems is that in the existing systems, external components store the data in memory. In *SLR*, memory instances are declaratively defined which selectively store data from the input flow of events to the *SLR*.

The storage of data in *SLR* is similar to active memories such as the ones of *IDA* and *CAST* as data is recorded in memory instances with unique identifiers, however *SLR* supports logical reasoning over the contents of memory instances. This approach supports having different memory profiles for different pieces of data and a flexible way of selecting the data that are to be reasoned about as a whole, thus allowing to reason about a part of knowledge that could be inconsistent with other part of the knowledge maintained in the memory.

Furthermore, active memories allow external components to update the contents of memory instances. As such, suitable error handling and locking mechanisms are necessary to synchronize the parallel access to memory. In contrast, the modeling of the input as a stream of events and clear semantics of memory instances in *SLR* removes much

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

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

*SLR* supports notifying external components when memory instances are added or deleted to the memory. This can be easily extended to also generate corresponding notifications when events are added or deleted from memory instances. However, the input flow of events to *SLR* is processed by *ETALIS*. Therefore external components can subscribe to *ETALIS* to be notified when the data of interest is being fed to *SLR*. While notifying changes of the memory is a main functionality in active memories, it is less common in logic-based knowledge management systems. An exception is *ORO* to which one can subscribe to receive a notification, whenever a fact can be inferred by the *ORO* knowledge base. However it is not described whether or not this includes the knowledge that can be derived by *SWRL* rules. Moreover, it not described whether this functionality is implemented by continuously querying the knowledge base for such a fact, or it is efficiently realized by an incremental and event-driven algorithm such as backward chaining rules in *ETALIS* [Anicic et al., 2012, 2010, Anicic, 2011].

## 4.4 Summary

*SLR* language is developed in this thesis to address the on-demand processing requirements related to discreteness, continuity and asynchronicity of robotic sensory information. *SLR* is a *Prolog*-based language extending the capabilities of existing logic-based robotic knowledge management systems with active memory functionalities. In this way, *SLR* unifies, advances and complements the capabilities of the state-of-the-art robotic on-demand processing systems. This is shown by presenting a detailed comparison of *SLR* with existing systems and approaches.

The input to *SLR* is modeled as a stream of events. *SLR* provides a high-level syntax to specify what parts of this stream should be maintained in the knowledge base. Records of data are maintained by memory instances, selectively recording parts of the input stream filtered by types of events and logical reasoning on their contents. These

memory instances are attached time-based and count-based memory profiles automatically removing outdated data. *SLR* supports active memory notification by generating corresponding events when memory instances are updated. These events can then be processed by *Etalis* part of *Retalis* to detect patterns of changes of the knowledge base to inform external components. In addition, *SLR* provides syntax support for state-based representation of knowledge built upon discrete events and also implements two mechanisms to synchronize queries which are to be answered based on events received asynchronously. *SLR* uses particular data structures and indexing mechanisms to efficiently implement its functionalities. An extensive empirical performance evaluation of *SLR* is presented in Chapter 5.

# Chapter 5

# Retalis Performance

Software development is inherently of high importance for this thesis as its aim is the support of timely processing, management and querying of information. In previous chapters, we presented the *Retalis* language providing a high-level syntax and clear semantics for implementing the on-flow and on-demand information functionalities of autonomous robots. The main key to usefulness of any information engineering language such as *Retalis* is its efficiency in execution of the processing, management and querying functionalities. This chapter empirically evaluates the performance of *Retalis* by demonstrating the implementation of an application for the *NAO* robot.

The remainder of this chapter is organized as follows. First, an implementation of an application for the *NAO* robot is presented. The performance of the *Retalis* language, integrating the *ELE* and *SLR* languages, as a whole is evaluated on this application which involves processing, management and querying of a flow of sensory information received at the rate of about 1900 events per second. We then present separate empirical performance evaluations of various information engineering functionalities of *Retalis* including memorizing, forgetting, querying and query synchronization.

## 5.1 NAO application

In this section, we report the development of an application for NAO robot using *Retalis* and *ROS*.[1] NAO is a small programmable humanoid robot offered by Aldebaran Robotics[2], equipped with advanced sensors such as cameras, touch sensors and microphones. In the application, NAO observes objects in the environment, perceiving their

[1] Appendix A presents a for a tutorial about the implementation of this application.
[2] http://www.aldebaran.com/en

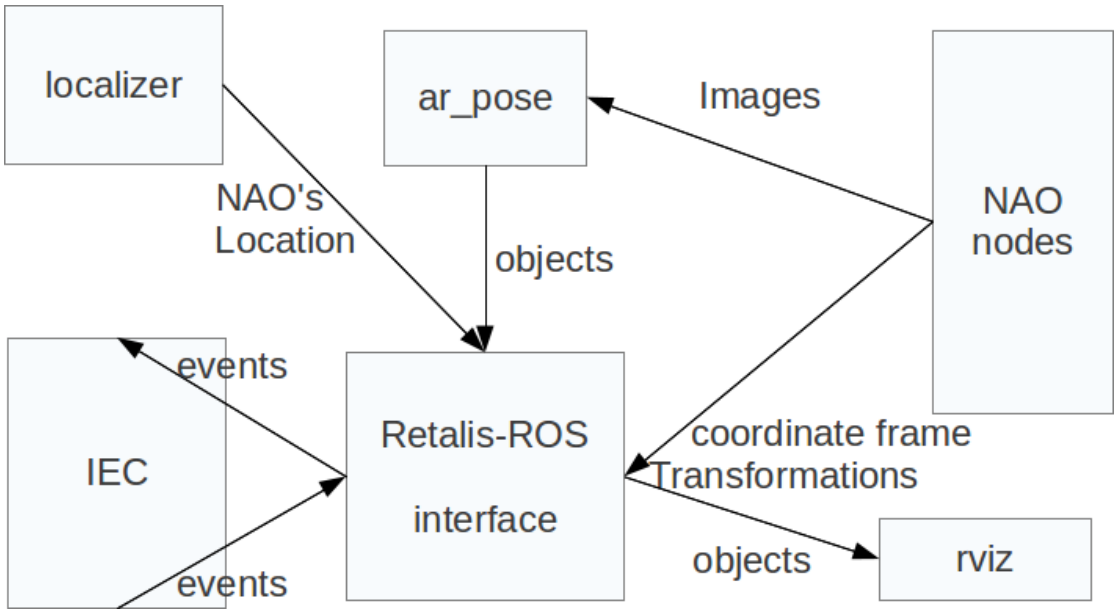


FIGURE 5.1: NAO's software components

relative positions to its camera, and computes the position of objects in the environment. Figure 5.1 presents software components[3] of the NAO application, operating as follows. The *NAO nodes*[4] component provides an interface to acquire sensory data and to command the NAO robot. It publishes images generated by the top camera of the robot. It also publishes events about the transformation among the robot's coordinate frames. Each of these events contains a set of transformations where each transformation specifies the relative position among two coordinate frames. The *ar_pose*[5] component processes the images to recognize objects and calculates the position of objects with respect to the camera. Each event from *ar_pose* contains data of a set of observed objects. The *localizer* component calculates the robot's position in the world. The *IEC* component is subscribed to information about objects' positions, robot's location and coordinate transformations. It calculates the position of objects in the world from the transformation among the following pairs of coordinate frames, *(world, base_link), (base_link, torso), (torso, neck), (neck, camera) and (camera, object)*. The arithmetic operations are performed using the *OpenGL Mathematics*[6] (GLM) library which has been integrated in *Retalis*. The *rviz*[7] component visualizes the objects in the environment. The *IEC* communication with other nodes is realized by the *Retalis-ROS interface* component. This component converts *ROS* messages to *Retalis* events and vice versa. The *IEC* and the *Retalis-ROS interface* components are implemented in *Retalis*.

[3]The software includes also a face recognition component which is not discussed for brevity.
[4]http://wiki.ros.org/nao_robot
[5]http://wiki.ros.org/ar_pose
[6]http://glm.g-truc.net/0.9.5/index.html
[7]http://wiki.ros.org/rviz

## 5.2 Evaluation

For a first test implementation, all software components run remotely on an XPS Intel Core i7 CPU@ 2.1 GHz x 4 laptop running ubuntu 12.04 LTS, connected to the NAO robot. After the evaluation phase, the software will be implemented in the NAO robot itself. NAO comes with an Intel Atom CPU@1.6 GHz running Linux. The performance is evaluated by measuring the CPU time, the amount of time of a CPU of the computer that is used by the *Retalis* program. We measure the CPU time as the percentage of the CPU's capacity (i.e. CPU usage percentage) computed by the operating system. In the following graphs, the vertical axis represents the CPU usage percentage and the horizontal axis represents the running time in seconds. The CPU time is logged every second and is plotted using "gnuplot smooth bezier".

The NAO application includes the following tasks:

- On-flow processing: events from *ar_pose* and *NAO nodes* are split into respective events such that each event contains data of a single object or the transformation among a single pair of coordinate frames. The transformation data among pairs of coordinate frames are published with frequencies from 8 to 50 hertz. There are in average 7 objects perceived per second. In total, *Retalis* processes about 1900 events per second.
- Memorizing and forgetting: there are 5 memory instances observing the events. They record and maintain the last 30 seconds histories of the transformation among the pairs of coordination frames used to calculate the transformation among *world* and *camera*.
- Querying memory instances: for each observed object, *SLR* is queried for the *world-to-camera* transformation. The transformation among a pair of coordinate frames at a time is calculated by interpolation, as performed by the rule (17) in page 18. Each interpolation requires accessing a memory instance twice, once using a *prev* term and once using a *next* term. To calculate the position of all objects, memory instances are accessed 70 times per second.
- Synchronization: a query is delayed in case any of the necessary transformations can not be interpolated from the data received so far. *Retalis* monitors the incoming events and performs the delayed queries as soon as all data necessary for their evaluations are available.
- Subscription: there are 8 distinct objects in the environment and consequently 8 subscriptions to publish recognized objects to distinct *ROS* topics. The *rviz* component is subscribed to these topics to visualize the position of objects.

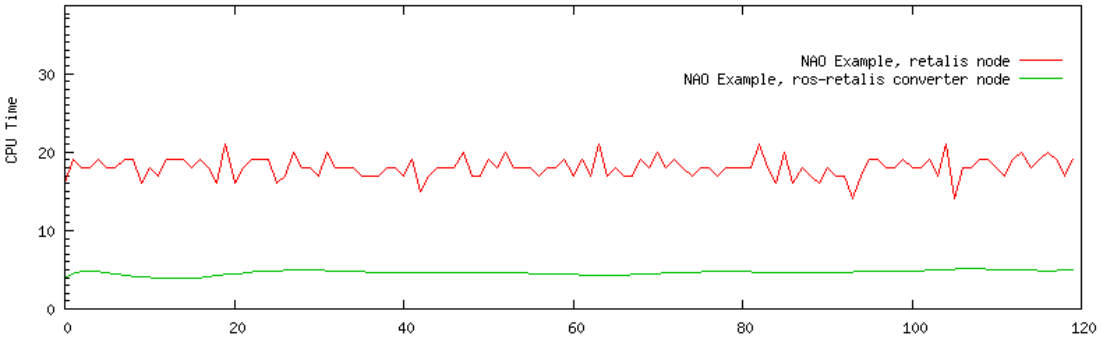


FIGURE 5.2: NAO application

Figure 5.2 shows the CPU time used by the *Retalis* and *Retalis-ROS*-converter nodes when running the NAO application. The *Retalis* node calculates the position of objects in real-time. It processes about 1900 events, memorizes 130 new events and prunes 130 outdated events per second. It also queries memory instances, 70 times per second. These tasks are performed using about 18 percent of the CPU time. In this experience, the *Retalis* node has been directly subscribed to *ROS* messages containing information about coordinate transformations and recognized objects. The *Retalis-ROS*-converter, consuming about 5 percent of CPU time, only subscribes *Retalis* to the recognized faces and converts and publishes events about objects' positions to *ROS* topics.

As we saw in Chapter 2, *Retalis* provides an easy way to subscribe to *ROS* topics and automatically convert *ROS* messages to events. This is implemented by the *Retalis-ROS*-converter node. The implementation is in Python and is realized by inspecting classes and objects at runtime and therefore is expensive. Figure 5.3 shows the CPU time used by the *Retalis* and *Retalis-ROS*-converter nodes for the NAO application, when the *Retalis-ROS*-converter is used to convert all *ROS* messages to *Retalis* events. In the previous configuration, the conversion from *ROS* messages, containing information about coordinate transformations and recognized objects, to events was performed by a manually written *c++* code, rather than using the *Retalis* automatic conversion functionality written in *Python*. We observe that in the new configuration, the *Retalis* node consumes a few percent less, but the *Retalis-ROS*-converter node consumes about forty percent more CPU time, comparing to the previous configuration. These results show that while the automatic conversion among messages and events are desirable in a prototyping phase, the final application should implement it in C++ for performance reasons. We will investigate the possibility to optimize and re-implement the *Retalis-ROS*-converter node in C++.

Metric evaluation of languages and systems like *Retalis*, in general, is challenging for the following reasons[Lemaignan, 2012, Langley et al., 2009, Mavridis and Deb Roy, 2006]. Experiments often involve many other modules running in parallel and building

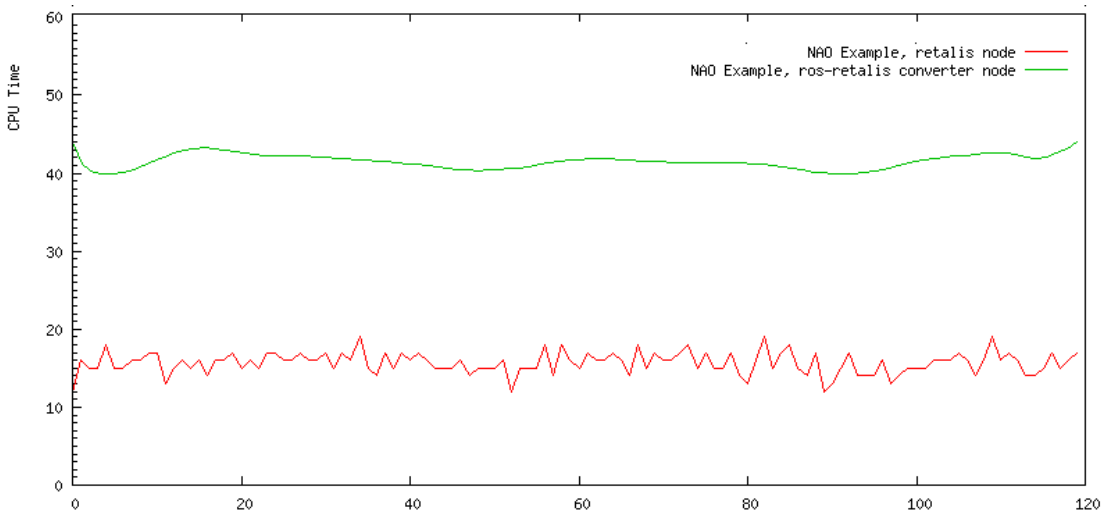


FIGURE 5.3: NAO application with automatic conversion of messages and events

repeatable experiments for robots in dynamic environments is challenging. In addition, very few existing systems report metric evaluations and the lack of standard *API*s and differences in functionalities makes it hard to compare these systems. The rest of this chapter evaluates main *Retalis* functionalities. We report a number of experiments using data from the NAO application, recorded by *rosbag*.[8] Using *rosbag*, data can be played in a simulation, as if it is played in real-time. While single performance results in the following experiments depend on the NAO application, a serie of experiments is presented for each functionality, allowing us to make a number of general observations about the performance of *Retalis* functionalities.

### 5.2.1 Forgetting and Memorizing

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

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

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

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

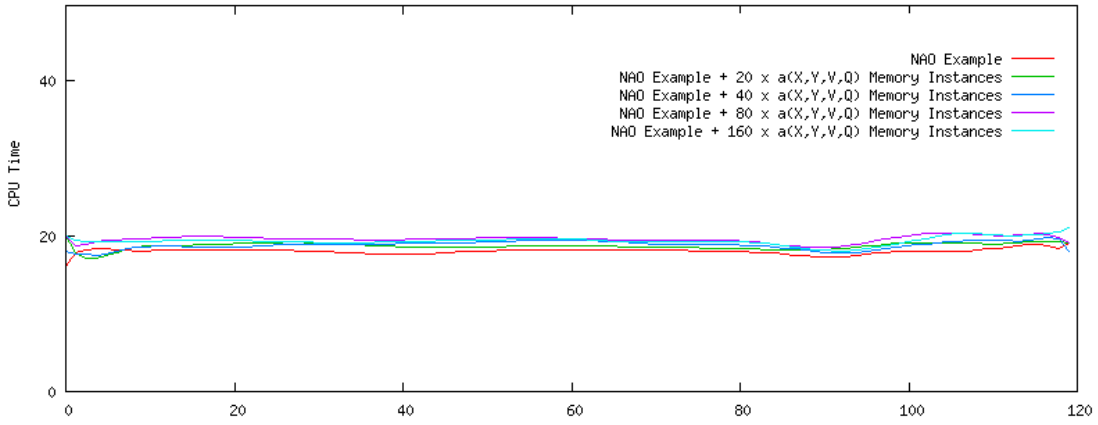


FIGURE 5.4: Irrelevant memory instances

Figure 5.4 shows the CPU time for a number of runs where up to 160 memory instances are added to the NAO application. These memory instances record *a(X,Y,Z,W)* events. Among the events processed by *Retalis*, there are no such events. The results show that the increase in CPU time is negligible. This shows that a memory instance consumes CPU time only if the input stream of events contains events whose type matches the type of events the memory instance records.

In Figure 5.5, the green and blue lines show the CPU time for cases where 20 memory instances of type tf(X,Y,V,Q) are added to the NAO application. These memory instances match all tf events, about 1900 of such is processed every second. The size of memory instances for the green line is 2500. These memory instances reach their size limit in two seconds. After this time, the CPU time usage is constant over time and includes the costs of unification, assertion and retraction for updating 20 memory instances with 1900 events per second. The size of memory instances for the blue line is 150,000. It takes about 80 seconds for this memory instances to reach their size limit. Consequently, the CPU time before the time 80 only includes the costs of unification and assertion, but not the costs of retraction. After the time 100, the CPU usages of both runs are equal. This shows that the cost of a memory instance does not depend on its size.

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

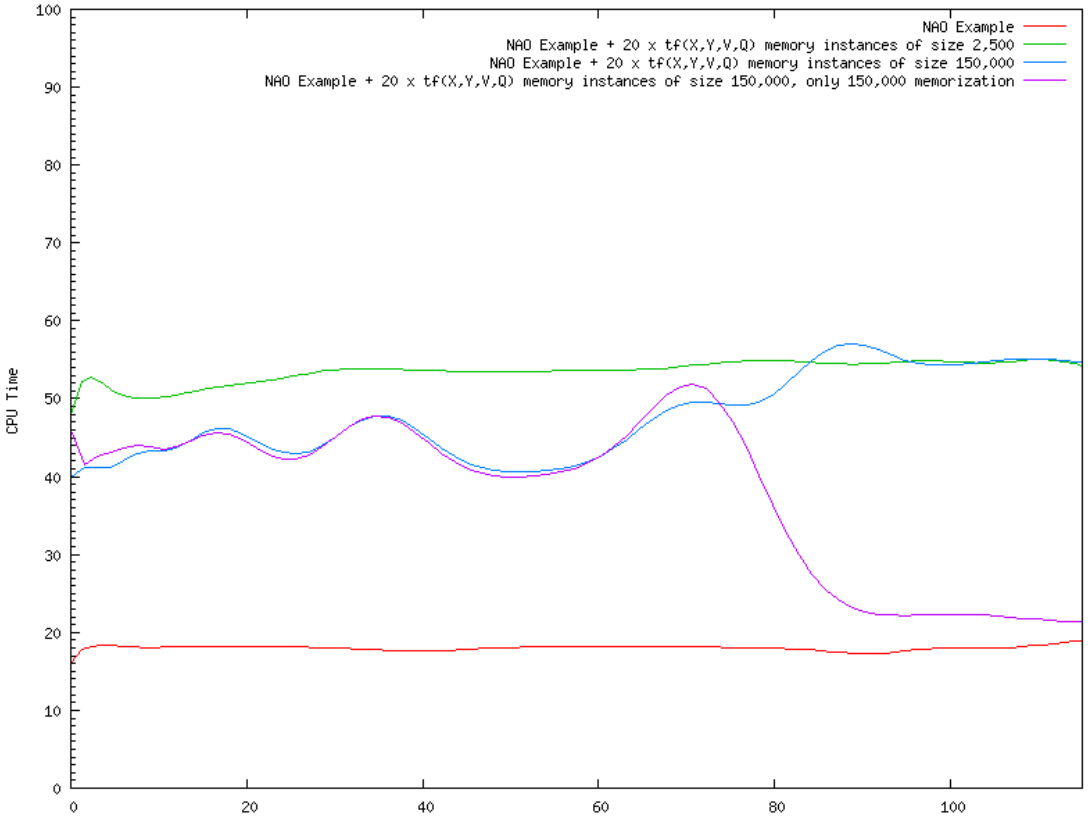


FIGURE 5.5: tf(X,Y,V,Q) memory instances (1)

memory updates (i.e. assertions or retractions) are processed using one percent of CPU time.

Figure 5.6 shows the CPU time for a number of runs where up to 40 memory instances of type tf(X,Y,Z,W) and size 2500 are added to the NAO application. The red line at the bottom shows the CPU time for the NAO application. We make the following observations. Adding first 10 memory instances to the NAO application increases the CPU time about 20 percent. After that, adding each set of 10 memory instances increases the CPU time about 13 percents. This shows that the cost grows less than linearly. The implementation of memory instances is in a way that the cost of an assertion or a retraction can be assumed constant. This means that the unification cost for the first set of memory instances is the highest. In other words, the unification cost per memory instance decreases when the number of memory instances are increased. The reason relates to the way that the underlying *SWI-Prolog* engine searches and unifies terms which is not investigated here.

Figure 5.7 shows the CPU time for a number of runs where up to 640 memory instances of type tf(head,camera,Z,W) and size 2500 are added to the NAO application. The events matching these memory instances are received with the frequency of 50 Hz. We make the following observations. First, it takes 50 seconds for these memory instances

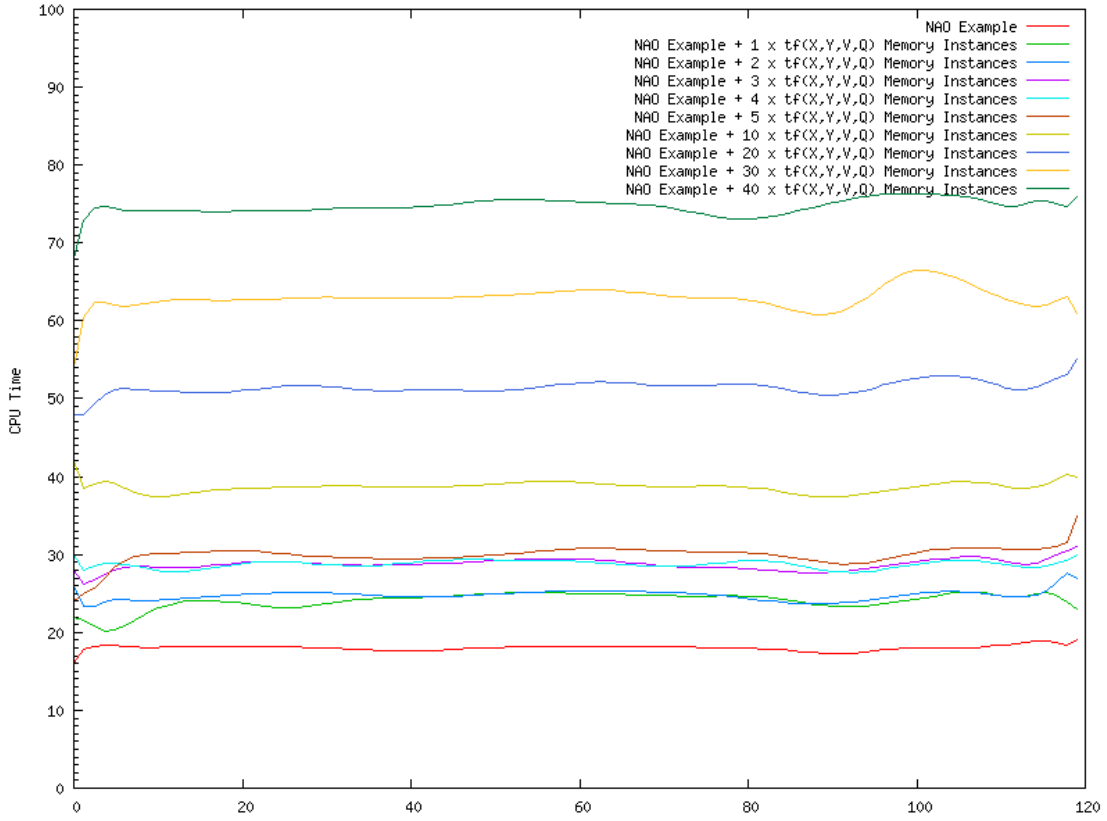


FIGURE 5.6: tf(X,Y,V,Q) memory instances (2)

to reach their size limit. After 50 seconds, these memory instances reach their maximum CPU usages, as the costs of retraction is added. Second, each memory instance filters 1900 events per second recording about two percent of them. The cost of 640 memory instances is about 35 percent of CPU time. Third, the unification cost per memory instance is decreased when the number of memory instances are increased.

Figure 5.8 compares the costs of different types of memory instances. The purple line shows the CPU time for the case where there are 10 memory instances of type tf(X,Y,V,Q). The green line shows the CPU time for the case where there are 320 memory instances of type tf(head,cam,V,Q). We observe that the costs of both cases are equal. The memory instances in the former case record 19,000 events per second (i.e. 10*1900). The memory instances in the latter case filter 1900 events per seconds for tf(head,cam,V,Q) events, recording 16000 events per second (i.e. 320*50). The results show the efficiency of the filtering mechanism.

The brown line shows the CPU time for the case where there are 10 memory instances of type tf(X,Y,V,Q) and 320 memory instances of type tf(head,cam,V,Q). Comparing it with the green and purple lines shows that the CPU time usage of these memory instances is less than sum of the CPU usages by 10 tf(X,Y,V,Q) memory instances and 320 tf(head,cam,V,Q) memory instances. This shows that the unification cost per

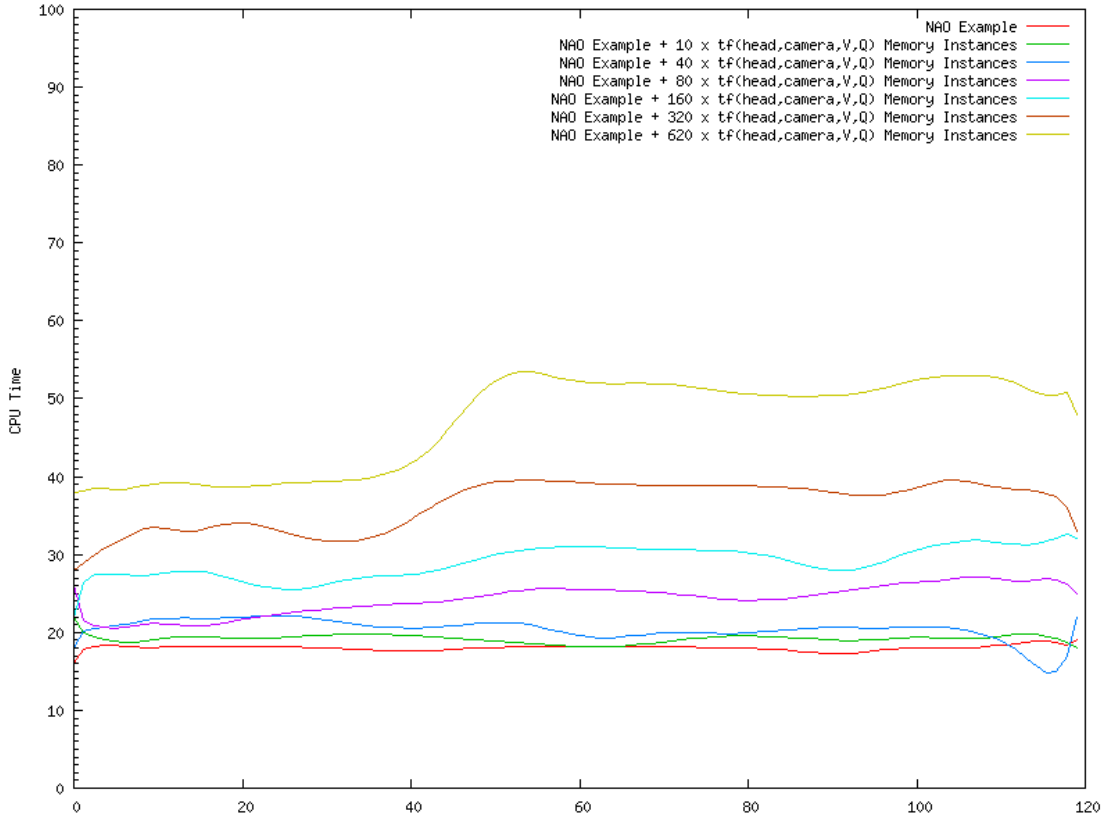


FIGURE 5.7: tf(head,cam,V,Q) memory instances

memory instance is decreased when the number of memory instances are increased, even when the memory instances are not of the same type.

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

### 5.2.2 Querying

*Retalis* queries are Prolog-like queries executed by the SWI-Prolog system. The following evaluates the performance of next and prev terms and the synchronization mechanism

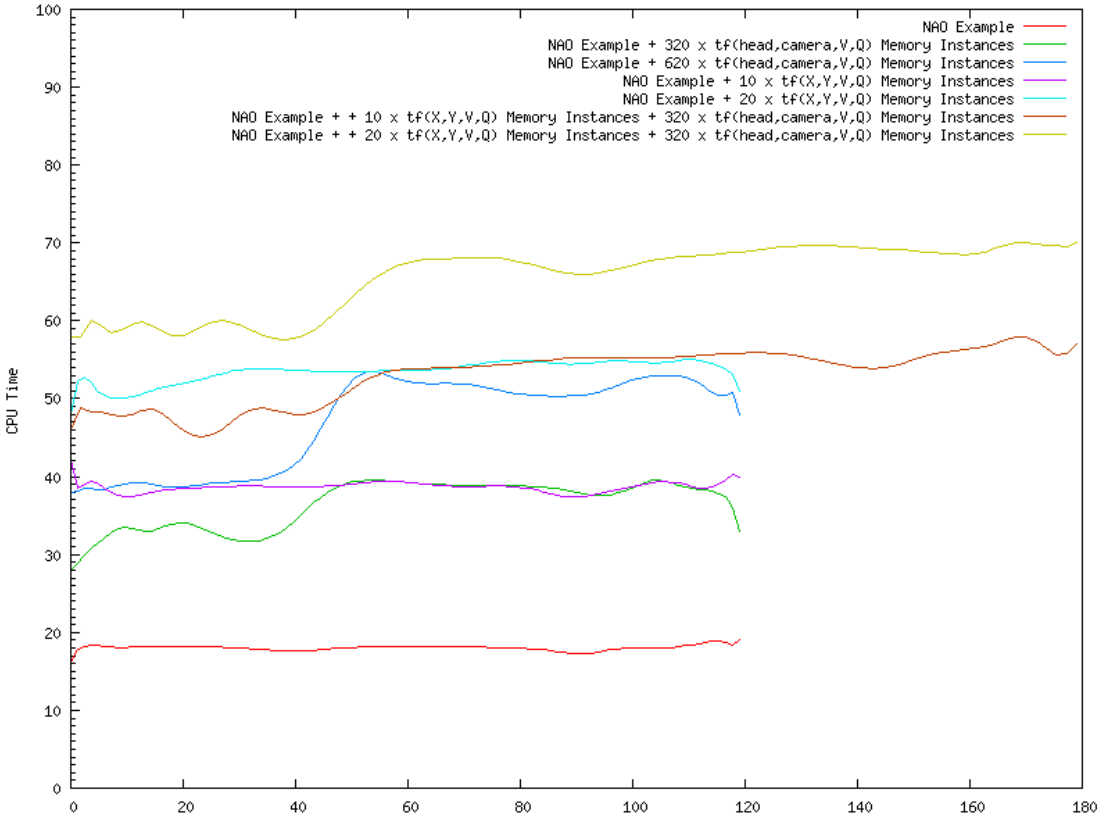


FIGURE 5.8: Memory instances of different types

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

### Querying Memory Instances

This section evaluates the performance of prev and next terms used to access event records in memory instances. *Retalis* optimizes the evaluation of these terms as follows. It keeps track of the number of event records in each memory instance. The prev and next terms are evaluated by a binary search on event records. An access to an event record by its number takes a constant time. Consequently, the evaluation of prev and next is done in logarithmic time on the size of the respective memory instance. In Figures 5.9, 5.10 and 5.11 below, the red line visualizes the CPU time of the NAO application.

The green line in Figure 5.9 visualizes the CPU time of the NAO application adapted as follows. There is an additional tf(head,cam,V,Q) memory instance of size 128. This memory instance is queried by 1000 next terms for each recognition of an object. In

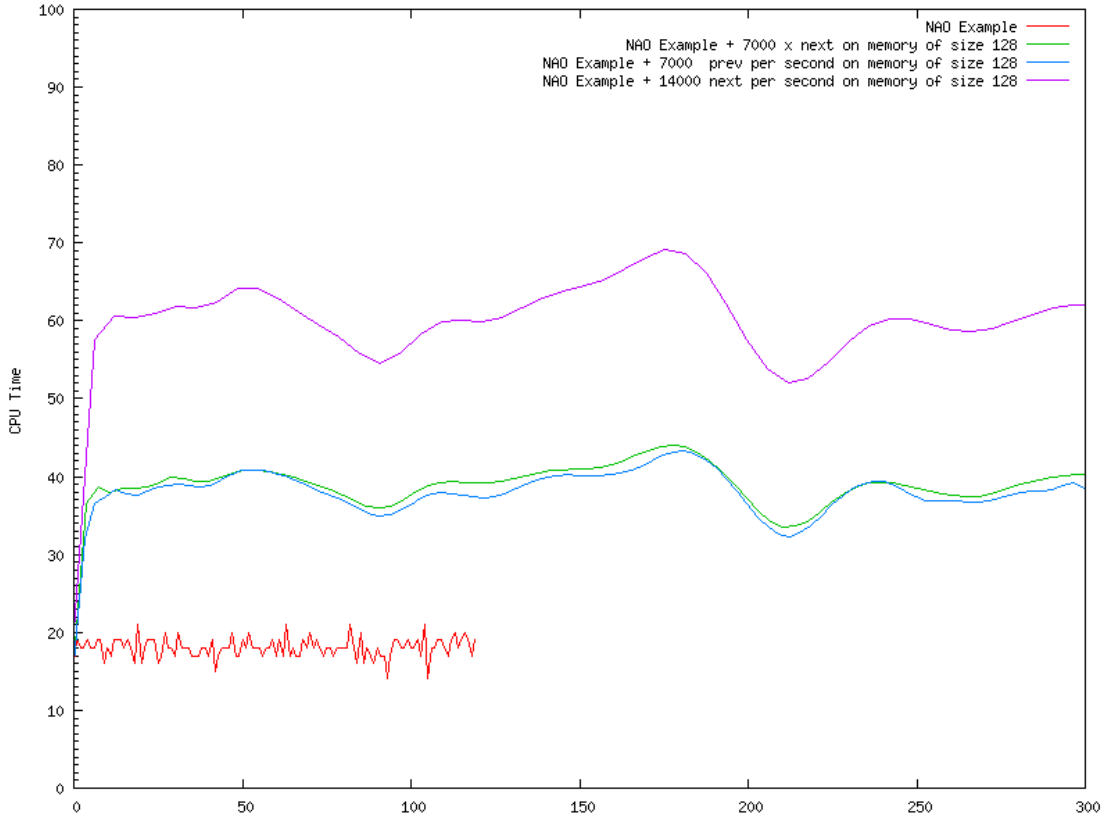


FIGURE 5.9: Next and prev terms (1)

average, 7000 next terms are evaluated per second. The blue line visualize the CPU time of a similar program in which 7000 prev terms are evaluated per seconds. The figure shows that the costs of the evaluations of prev and next terms are similar. The purple line shows the CPU time of the case where 14,000 next terms are evaluated per second. We observe that the cost grows linearly.

The blue line in Figure 5.10 visualizes the CPU time of the case where 7000 next terms are evaluated per second. The green line visualizes the CPU time of the case where there are 320 tf(head,cam,V,Q) memory instances added to the NAO application. The purple line visualizes the CPU time of the case where 7000 next terms are evaluated per second and there are 320 tf(head,cam,V,Q) memory instances. We observe that the cost of accessing a memory instance does not depend on existence of other memory instances.

The green line in Figure 5.11 visualizes the CPU time of evaluating 7000 next terms per second on a memory instance of size 128. The blue linevisualizes the CPU time of evaluating 7000 next terms per second on a memory instance of size 16384. The size of the memory instance in the latter case is the power of two of the size of the memory instance in the former case. The increase in the CPU time for the latter case, with respect to the NAO application, is less than two times of the increase in the CPU time for the former case.

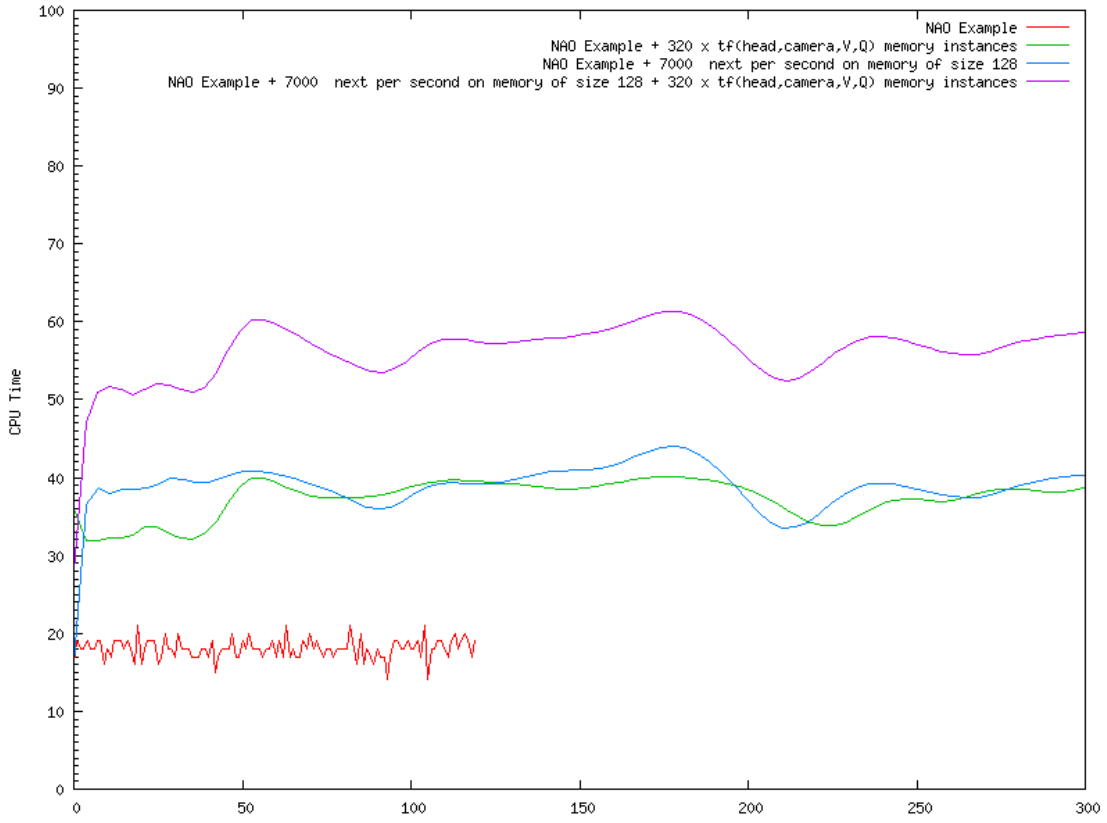


FIGURE 5.10: Next and prev terms (2)

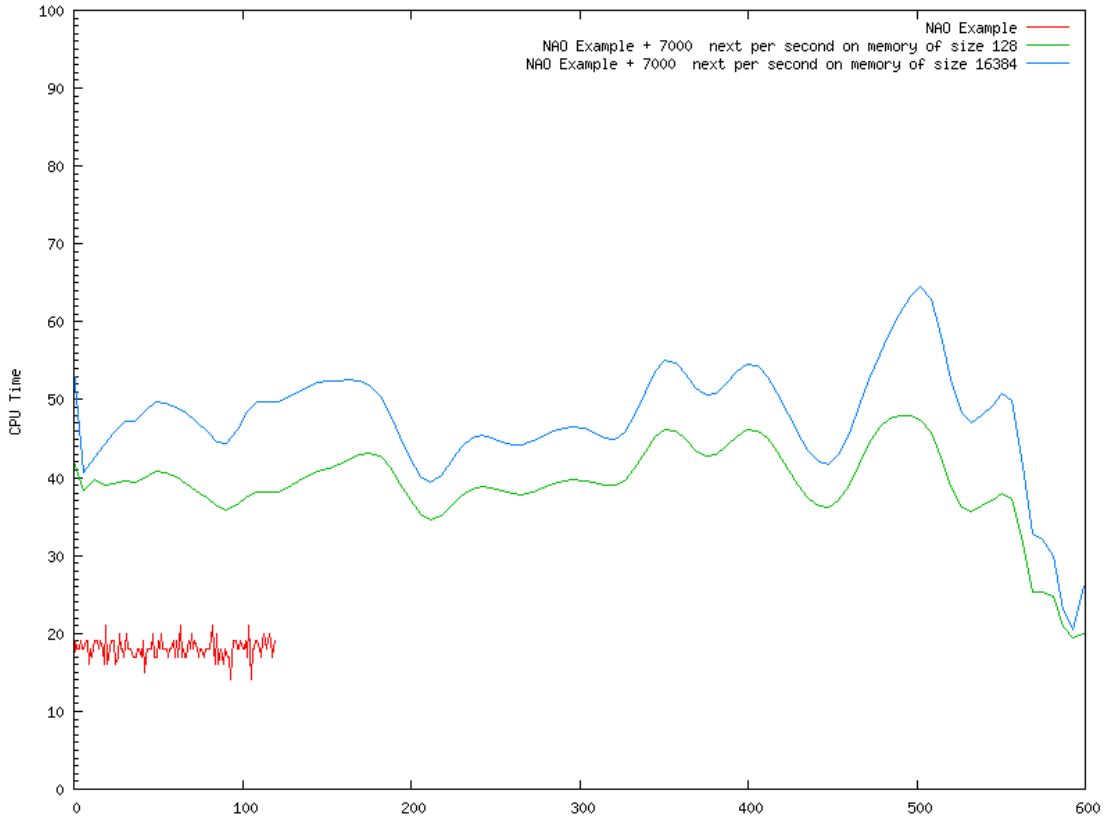


FIGURE 5.11: Next and prev terms (3)

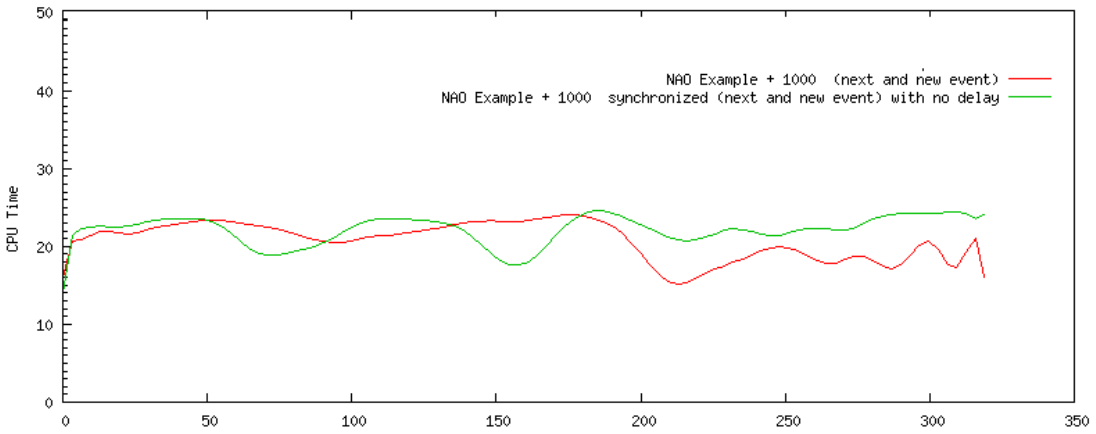


FIGURE 5.12: Synchronization with no delay

The *prev* and *next* terms provide efficient ways of accessing records of events. Otherwise, all event records should be read, for instance, to find the latest position of an object. For example, an experiment is reported for the *KnowRob* knowledge base where there are 65,000 records of events about the location of an object. It takes 11 seconds to find the latest location [Tenorth and Beetz, 2012].

### Synchronization

The synchronization mechanism is implemented as follows. Before evaluating a query, memory instances are checked whether they are up-to-date with respect to the query (i.e. the query is achievable as defined in Section 4.2.5). If the query cannot be evaluated, it is recorded as a postponed query. For each postponed query, *Retalis* generates a set of monitors. Monitors observe memory update events. As soon as all necessary events are in place in memory instances, the query is performed. The implementation of monitors are similar to the implementation of memory instances.

The red line in Figure 5.12 visualizes the CPU time of the NAO application where in each second, 1000 next queries on a memory instance of size 2500 are evaluated. In addition, for each next query, a new event is generated. The green line visualizes the CPU time of a similar case where the next queries are synchronized. This experiment is conducted in a way that no query needs to be delayed. Comparing these two cases shows that when queries are not delayed, the synchronization cost is negligible.

Figure 5.13 shows the CPU time of four cases. In all these cases, 1000 synchronized next queries are evaluated and 1000 events are generated in each second. The red line visualizes the case where no query is delayed. The green line visualizes the case where queries are delayed for 5 seconds. In this case, the memory instance queried by a next term has not yet received the data necessary to evaluate the query. The query is

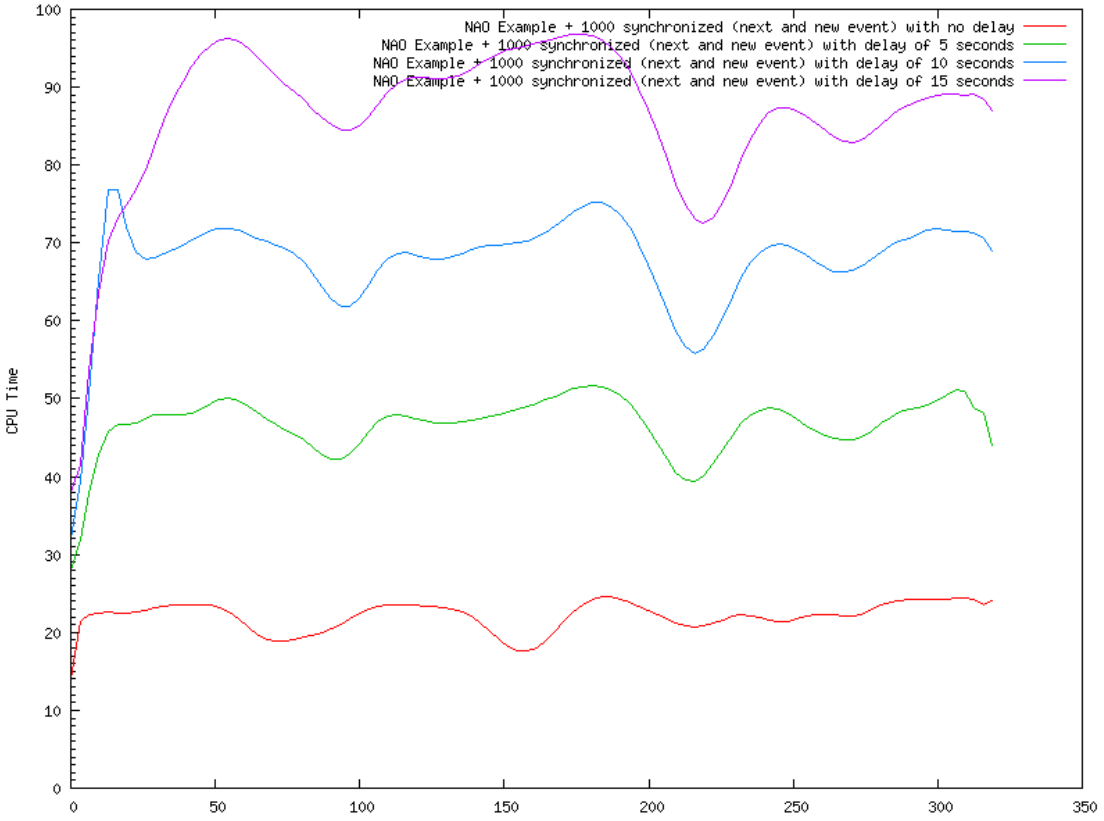


FIGURE 5.13: Synchronization with delays

performed as soon as the memory instance is updated with relevant information. There are 1000 queries per seconds, each delayed for 5 seconds. This means there exist 5000 monitors at each time. These monitors observe 1900 events processed by *Retalis* per second. We observe that for such a large number of monitors observing such a high-frequency input stream of events, the increase in CPU time is less than 30 percent.

### 5.2.3 On-Flow Processing

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

Information flow processing systems such as *Etalis* are designed for applications that require a real-time processing of a large volume of data flow. We refer the reader to the evaluation of the performance of *Etalis* presented elsewhere [Anicic et al., 2012, Anicic, 2011]. The evaluation shows, in terms of performance, *Etalis* is competitive with respect to the state-of-the-art information processing systems.

### 5.2.4 Subscription

The implementation of the subscriptions is similar to the implementation of memory instances. The only difference is that an event matching a memory instance is asserted to the knowledge base for that memory instance, and an old event is retracted if the memory instance is full, but an event matching a subscription is delivered to the respective subscriber. Consequently, the costs of subscriptions include the unification cost, discussed in section 5.2.1, and the costs to publish events to subscribed *ROS* topics. The latter comprises the costs for converting events to *ROS* messages and the costs of message transportation within the *ROS* framework.

## 5.3 Summary

Metric evaluation of languages and systems like *Retalis*, in general, is challenging. Experiments often involve many other modules running in parallel and building repeatable experiments for robots in dynamic environments is challenging. In addition, very few existing systems report metric evaluations and the lack of standard *API*s and differences in functionalities makes it hard to compare these systems.

An application for *NAO* robot is presented involving processing, management and querying of a large flow of sensory information, received at the rate of about 1900 events per second. The performance of *Retalis*, measured by *CPU* usage, is evaluated on this application. Furthermore, the performance of main functionalities of *Retalis* are separately evaluated in a number of experiments. The results show that *Retalis* can handle the processing of a large flow of events, filter it to selectively record a large amount of data in knowledge base, efficiently access the recorded data and efficiently synchronize the queries. Consequently, *Retalis* is proven to be an efficient language to implement the information engineering functionalities of autonomous robots.

# Chapter 6

# Active Queries

The question of this chapter is how to support incremental evaluation of logic program queries. Language support is needed to be able to register queries as *active queries* that are evaluated on the current knowledge base (i.e. logic program) and their results are incrementally and efficiently updated as the knowledge base changes. The question is how to provide an efficient mix of top-down (i.e. query-driven) and bottom-up (i.e. data-driven) query evaluation strategies deployed in on-demand and on-flow processing models. Active queries should be evaluated in a top-down manner and their results should be efficiently updated in a bottom-up manner until they are unregistered. The bottom-up evaluation strategy updating the query results should only take into account the queries which are active at the time.

The rest of this chapter is organized as follows. First, we describe the definite logic programs and the *Prolog* strategy to evaluate definite logic program queries. Then, we describe the *ELE* execution model for an incremental evaluation of definite logic programs. Afterwards, we present two approaches for implementation of active definite logic program queries, the Naive approach and the Optimized approach. The Naive approach builds directly on top of the *ELE* execution model to support registering and un-registering active queries at run-time. The optimized approach is a new approach developed in this thesis, revising the Naive approach to provide a more elegant and more efficient mechanism for implementation of active queries. The approach is further developed by incorporating the *tabling* technique to further optimize it and to deal with the problem of falling into infinite loops in bottom-up evaluation of logic programs that contain recursion. Finally, we present the related work and give a summary.

## 6.1 Introduction

Querying current logic programming systems such as *Prolog* follows the request-response pattern of interaction. An asker component submits a query and a logic programming system evaluates the query and responds with the results. Given a query, a top-down evaluation strategy is used examining the relevant rules and facts in the knowledge base to find answers of the query. The advantage is that the search is guided by the type of the query and its bound variables.

There are however cases where the same query should be continuously evaluated on the knowledge base. For example in *focused reasoning*, an asker wants to find an answer for a query and wants to find it as soon as there exists one [Schlegel and Shapiro, 2004]. If the knowledge base is a logic program and the query does not have an answer at the time, the asker should re-evaluate the query after each change of the knowledge base in order to find the answer as soon as it exists.

In current logic programming systems such as *Prolog*, re-evaluating the same query requires to re-submit the query which in turn results in re-computation of the query from scratch. Not only has this a disadvantage on performance but also on reactivity of the integrated system (e.g. robot). If the system's operations depend on monitoring results of some queries to its logic programming component, then the system has to repeatedly perform the queries. When the logic programming component is updated with new information, the system does not notice changes to results of the queries until the next time it re-evaluates them.

For example, BDI-based agent programs developed in agent programming languages [Bordini et al., 2006, Vikhorev et al., 2010] such as 2APL [Dastani, 2008], GOAL [Hindriks, 2009] and Jason [Bordini and Hübner, 2005, Píbil et al., 2012] contain a set of rules each generating a plan to reach a goal or to respond to an event, applicable in certain belief states. To determine the applicability of such rules, the same queries are repeatedly performed on the agent knowledge base resulting in performance issues [Alechina et al., 2012].

An approach to reduce the cost of repeated queries is to cache query results. When a query is re-evaluated, a caching mechanism determines whether the knowledge base has been updated with some facts which may be relevant to the query. If not, the cached results are used. For instance, such an approach has been recently adopted by a large portion of agent programming languages community [Alechina et al., 2013]. The limitation of this approach is that as soon as the knowledge base is updated with new information, the cached results of queries depending on such information are invalidated, hence those queries are to be re-evaluated from scratch. For example, when the fact *c(1)*

is added or deleted from the knowledge base, the cached results of the query *q(X,Y,Z)* is invalidated if the knowledge base contains the rule *q(X,Y,Z) :- a(X), b(Y), c(Z).* Moreover, this approach does not benefit from caching results of queries' sub-goals as it only caches results of *high-level* queries. For example, the two queries *q(2,1,Z)* and *q(1,1,Z)* evaluates their common sub-goal *b(1)* separately, the result of which could be re-used when computed once.

As opposed to backward reasoning systems where top-down query evaluation strategies are used, forward reasoning systems such as RETE [Forgy, 1982] and its descendant deploy bottom-up query evaluation strategies. Forward reasoning systems derive and keep an up-to-date list of all facts which can be inferred from the facts and rules in the knowledge base. In these systems, queries of interest are usually known in advance. Given a set of queries and a knowledge-base, the rules in the knowledge base are transformed such that the forward reasoning mechanism deployed is only concerned with the queries of interest. The knowledge-base then explicitly stores and maintains the up-to-date list of all relevant results that can be derived from the given facts and rules in the knowledge base. The advantages of these approaches is in their use of data-driven evaluation mechanisms to efficiently update the results of queries of interests, as facts are added or removed from the knowledge base.

This chapter in concerned with incremental evaluation of logic program queries to support the implementation of active queries mixing top-down and bottom-up evaluation strategies to take the advantages of both. From the software architecture point of view, an active knowledge base provides a means for other components to register logic programming queries as active queries, each assigned a unique id. As the knowledge base is updated with new information, it notifies changes to results of the registered queries to interested components. Such changes are the addition or deletion of some results for each query.

From the query evaluation point of view, an active query is evaluated on the current knowledge in the knowledge base in a top-down manner, but its results are incrementally and efficiently updated as the knowledge base is updated with new information. When updating query results in a bottom-up manner, the propagation of changes of the knowledge base is directed toward and constrained by those queries which are active at the time. Comparing to re-evaluation of queries from scratch, an incremental evaluation propagates the changes and thus only recomputes necessary parts of proof trees of queries.

## 6.2 Definite Logic Programs

A definite logic program consists of a set of logical clauses describing a problem domain. A clause or a rule is of the form $a \leftarrow a_1 \wedge .. \wedge a_n$. In a logic programming language, such clause is represented as $a$ :- $a_1 \wedge .. \wedge a_n$, the declarative meaning of which is $a$ is true, if $a_1$ and ... and $a_n$ are true. In such a rule, $a$ is called the *head* and $a_1 \wedge .. \wedge a_n$ is called the *body*. $a$ and $a_i$ are terms which are atomic formula 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. $a \leftarrow true$ is called a *fact* and is represented as $a$.

### 6.2.1 SLD Resolution

*Prolog* execution is query-driven. Given a query $Q$ to be evaluated on a definite logic program $P$[1], the *Prolog* engine uses the *SLD* resolution strategy to determine whether or not $Q$ is a logical consequence of $P$ [Lloyd, 1984b]. To explain this strategy, let's assume $Q$ is a single term. *Prolog* performs a top-down depth-first search to evaluate $Q$. The query is regarded as a goal and relevant rules and facts in the knowledge base (i.e. logic program) are tried to find answers of $Q$.

A goal is proven, if there is a variable substitution by applying which the goal matches a fact, or matches the head of a rule and the (sub-)goals in body of the rule can be proved from left to right. When matching the head of a rule, the variable substitution used is applied to the body of the rule before proving the goals in the body. In addition, the variable substitution used to prove a goal in the body is applied to all goals in the body in the right hand side of the goal before proceeding to the rest. When the query is derived from the program using this strategy, the variable substitution used to prove the goal is returned to the user as an answer.

For each goal, there may be different facts or rules in the program matching it. In such a case, *Prolog* creates a choice-point and unifies the goal with the first alternative. It then continues with its execution until it proves the query or a goal fails. In the latter case, *Prolog* backtracks by disregarding all variable bindings made since the most recent choice-point and trying the next alternative of that choice-point. When choosing among alternatives, rules and facts are always tried in the order they appear in the logic program. Backtracking is performed in the case of a failure, but It is also used to find other ways of proving a goal.

[1] *Prolog* syntax extends the definite logic program syntax with built-in predicates and language operators such as the *negation as failure* predicate and the *cut* operator and employs the *SLDNF* resolution method to evaluate the query [Apt and van Emden, 1982]. In this chapter, we only consider definite logic programs.

For an example, consider the Program 1 presented in Listing 6.1. *Prolog* evaluates the query *a(X,Z)* on this program as follows. The goal *a(X,Z)* matches the head of the first clause, resulting in evaluation of the goals in body of the clause from left to right. The first goal *b(X,Y)* is resolved using the *b(1,2)* clause, binding the variables $X$ and $Y$ to 1 and 2, respectively. The execution continues with evaluating the second goal in the body, *c(Y,Z)*. As $Y$ has been bound to 2, the goal to be evaluated is *c(2,Z)*. This goal matches the second and fourth clauses, creating a choice-point. The first alternative to resolve *c(2,Z)* is to use the second clause, producing the new goal *e(2,Z)* which is resolved against the *e(2,4)* clause, binding $Z$ to 4. Then, the third goal *d(4)* is evaluated but does not match any clause. Consequently, this path of execution fails and the execution backtracks to the last choice point. The second alternative to resolve *c(2,Z)* is using the *c(2,3)* clause, binding $Z$ to 3. Then, the goal *d(3)* is evaluated, matching the fifth clause. At this point, all goals have been resolved, the query has been proven and the answer *X=1, Z=3* is returned. As all choice-points have been exhausted, no more answer can be derived and the execution terminates.

```
1 a(X,Z) :- b(X,Y), c(Y,Z), d(Z).
2 c(Y,Z) :- e(Y,Z).
3 b(1,2).
4 c(2,3).
5 d(3).
6 e(2,4).
```

Listing 6.1: Program 1

In *Prolog*, the knowledge, encoded as a logic program, can be updated by addition or deletion of facts using the *assert* and *retract* meta-predicates, respectively. For example, suppose that the clause "*add(d(X)) :- assert(d(X)).*" is included in Program 1. Now if the query *add(d(4))* is evaluated on Program 1, the goal *add(d(4))* is called matching the head of the "*add(d(X)) :- assert(d(X)).*" clause, binding $X$ to 4. Consequently, the goal *assert(d(4))* in the body is called, adding the fact *d(4)* to Program 1. As all goals has been resolved, the query *add(d(4))* succeeds. This example shows the evaluation of a *Prolog* query can have side effects such as adding a fact to the knowledge base. Alternatively, we could directly add *d(4)* to Program 1 by executing the query *assert(d(4))*.

Suppose that the knowledge base (i.e. Program 1) is updated by addition of the fact *d(4)*. If the query *a(X,Z)* is an active query, then we need to re-evaluate it to check whether its results have been changed by the update. When the query is re-evaluated on the updated knowledge base, the answers *X=1, Z=4* and *X=1, Z=3* are derived.

The query evaluation steps are similar as before except that the goal *d(4)* does not fail but matches the newly added fact *d(4)*, deriving the answer *X=1, Z=4*. This chapter develops the Naive and Tabled Optimized approaches that efficiently derive the answer *X=1, Z=4* from the addition of *d(4)*, without the need for re-evaluation of the query from scratch. Suppose that now the knowledge base is updated by removal of the fact *d(3)*. Again, we need to re-evaluate the query *a(X,Z)* which results only in the answer *X=1, Z=4*. When the fact *d(3)* is removed, the Naive and Tabled Optimized approaches efficiently compute that the answer *X=1, Z=3* should be removed from the set of answers of the query *a(X,Z)*, without the need for re-evaluation of the query from scratch.

## 6.3 ELE Execution Model

As described in Chapter 3, an *ELE* program consists of a set of event rules, specifying events to be detected in terms of patterns of other events. An *ELE* program can be viewed as a definite logic program where *ELE* event rules are *Horn* clauses and events are ground facts. In fact, the *ELE* execution system first parses the event rules into *Horn* clauses [2] before applying further transformations on them. However, the execution model of *ELE* is data-driven as opposed to query-driven execution model of *Prolog*.

In *Prolog*, each rule $a$ :- $a_1 \wedge .. \wedge a_n$ is interpreted as "to prove $a$, prove $a_1 \wedge .. \wedge a_n$", but in *ELE*, such a rule is interpreted as "if $a_1 \wedge .. \wedge a_n$ is proven, then $a$ is proven. In other words, *Prolog* is asked a query and *SLD* resolution is used to answer the query using facts and rules in the knowledge base, but *ELE* derives all facts that can be inferred from a given set of rules and facts and there is no explicit notion of query.

There is also another difference between *Prolog* and *ELE* programs. A *Prolog* program contains a set of facts and rules. When the program is updated, for instance, by addition of a fact using the *assert* predicate, the fact becomes part of the program. In contrary, the set of rules of an *ELE* program does not contain any fact. An *ELE* program receives facts as its input. When a fact is fed to an *ELE* program as its input, we say that the fact is added to the program. However, the fact does not become a first class citizen of the program as in *Prolog*, but it is processed in order to derive new facts according to the *ELE* rules. Consider $P$ and $F$ to be an *ELE* program (i.e. a set of *Horn* clauses containing no facts) and a set of facts, respectively. If the facts in $F$ are added to $P$ (i.e. processed as inputs of $P$), then the *ELE* execution system derives all facts that can be inferred given the logic program $P \cup F$. In general, the *ELE* execution system sends the derived facts to external components subscribed to such facts and disregards them

[2] The transformed program may include *negation as failure* but we only consider definite logic programs.

afterwards. However, some derived facts are memorized by *ELE* to possibly derive other facts using future input facts.

As for addition, the notion of deletion for *ELE* is different from *Prolog*. When a fact is added to *ELE*, *ELE* derives all facts that can be inferred using the newly added fact in combination with the *ELE* rules and previous input facts. External components are then notified about the derived facts. When a fact is deleted from *ELE*, *ELE* computes all facts that were derived using the deleted fact and notifies the external components about those facts.

Given an original program $P_o$, a definite logic program containing no fact, the *ELE* execution system transforms it into a logically equivalent *Prolog* program $P_t$ (i.e. transformed program), whose execution model is data-driven. The transformation is such that as new facts are processed by the system, intermediate goals are generated, progressing toward detection of derived facts and a derived fact is detected as soon as the last fact required for its detection is processed by the system. When facts are deleted, dependent intermediate goals are deleted and dependant derived facts are efficiently determined as deleted to notify the external components.

For an example, consider the first two clauses of program 1 to be the *ELE* program $P_o$ and suppose that the external component *comp1* is subscribed to *ELE* for the facts of the form *a(X,Y)*. The *ELE* execution system transforms $P_o$ to $P_t$, the Prolog program presented in Listing 6.2.[3] The *forall* clauses implement some loops. For instance, the *forall* clause in the second rule is read as follows. For all facts in the transformed program matching *goal(b(X,Y),c(Y,Z),$e_{12}$(X,Y,Z))*, call *add($e_{12}$(X,Y,Z))*. The set of such facts in the transformed program is initially empty. A fact $f$ is added to *ELE* by calling *add(f)* on the transformed program and applying all applicable rules. The transformed program contains also rules to implement a set of procedures for deletion of facts which have been omitted for brevity.

Now, suppose that the facts *b(1,2)*, *d(3)*, *c(2,3)* and *e(2,4)* are added to *ELE* in the corresponding sequence order. The execution of the transformed program is as follows. First, $b(1,2)$ is added by calling *add(b(1,2))* matching the head of the first two rules in $P_t$, binding $X$ to 1 and $Y$ to 2 in both cases. The first rule asserts the fact *goal(c(2,Z),b(1,2),$e_{12}$(1,2,Z))* into $P_t$. The execution of the second rule has no effect as the first argument of *forall* does not match any fact. Then, *d(3)* is added by calling *add(d(3))*, matching the seventh and eighth rules. Similarly, the effect is the assertion of the fact *goal($e_{12}$(X,Y,3),d(3),a(X,3))* into $P_t$. Then, *c(2,3)* is added by calling *add(c(2,3))*, matching the third and fourth rules, binding $Y$ to 2 and $Z$ to 3 in both cases. The third rule asserts the fact *goal(b(X,2),c(2,3),$e_{12}$(X,2,3))* into

[3]The code is not exactly the same as is produced by *ELE*.

$P_t$. Applying the fourth rule and given $Y$ and $Z$ respectively bound to 2 and 3, the *goal(c(2,3),b(X,2),$e_{1,2}$(X,2,3))* argument of *forall* matches *goal(c(2,Z),b(1,2),$e_{12}$(1,2,Z))* in $P_t$, binding $X$ to 1. Consequently, *add($e_{12}$(1,2,3))* is called, matching the the fifth and sixth rules, binding $X$, $Y$ and $Z$ respectively to 1, 2 and 3. The fifth rule asserts *goal(d(3),$e_{12}$(1,2,3),a(1,3)* into $P_t$. Applying the sixth rule, the *goal($e_{12}$(1,2,3),d(3),a(1,3))* argument of *forall* matches the fact *goal($e_{12}$(X,Y,3),d(3),a(X,3))* in $P_t$, calling *a(1,3)*. The call matches the 10th clause and the fact *a(1,3)* is sent to *comp1* to be added to the set of facts matching the subscription.

```
1  add(b(X,Y)): assert( goal(c(Y,Z),b(X,Y),e12(X,Y,Z)) ).

2  add(b(X,Y)): forall( goal(b(X,Y),c(Y,Z),e12(X,Y,Z)),
                        add(e12(X,Y,Z)) ).

3  add(c(Y,Z)): assert( goal(b(X,Y),c(Y,Z),e12(X,Y,Z)) ).

4  add(c(Y,Z)): forall( goal(c(Y,Z),b(X,Y),e1,2(X,Y,Z)),
                        add(e12(X,Y,Z)) ).

5  add(e12(X,Y,Z)): assert( goal(d(Z),e12(X,Y,Z),a(X,Z)) ).

6  add(e12(X,Y,Z)): forall( goal(e12(X,Y,Z),d(Z),a(X,Z)),
                            add(a(X,Z)) ).

7  add(d(Z)): assert( goal(e12(X,Y,Z),d(Z),a(X,Z)) ).

8  add(d(Z)): forall( goal(d(Z),e12(X,Y,Z),a(X,Z)),
                      add(a(X,Z)) ).

9  add(e(Y,Z)): add(c(Y,Z)).

10 add(a(X,Y)): notify('comp1','add',a(X,Y)).
```

LISTING 6.2: The transformed program $P_t$ for the first two clauses of Program 1

The example shows how the transformed program derives the fact *a(1,3)* in an incremental manner and derives it as soon as all necessary facts are added to *ELE*. Now, we add *e(2,4)* by calling *add(e(2,4))*, which matches the 9th rule calling *add(c(2,4))*. The call matches the third and fourth rules. The third rule asserts *goal(b(X,2),c(2,4),$e_{12}$(X,2,4))*

into $P_t$. Applying the fourth rule, the *goal(c(2,4),b(X,2),$e_{1,2}$(X,2,4))* argument of *forall* matches the fact *goal(c(2,Z),b(1,2),$e_{12}$(1,2,Z))* in $P_t$, binding $X$ to 1. Consequently, *add($e_{12}$(1,2,4))* is called, matching the the fifth and sixth rules, binding $X$, $Y$ and $Z$ respectively to 1, 2 and 4. The fifth rule assertz *goal(d(4),$e_{12}$(1,2,4),a(1,4))* into $P_t$ and the sixth rule has no effect.

Now, suppose that the fact *d(4)* is added to Program 1. To see the changes in results of the query *a(X,Y)* due to the update, we need to re-evaluate the query, when *Prolog* runs Program 1. When *Prolog* runs the transformed program of Program 1, we add *d(4)* by calling *add(d(4))*. The call matches the seventh rule asserting *goal($e_{12}$(X,Y,4),d(4),a(X,4))* into $P_t$. The call matches also the eighth rule where the *goal(goal(d(4),$e_{12}$(X,Y,4),a(X,4)* argument of *forall* matches the fact *goal(d(4),$e_{12}$(1,2,4),a(1,4))* in $P_t$, calling *a(1,4)*. Consequently, the new answer *X=1, Y=4* for the query *a(X,Y)* can be efficiently derived without the need for re-evaluation of the query from scratch.

### 6.3.1 Event-Driven Backward Chaining Rules

This section describes how *ELE* transforms a definite logic program into a *Prolog* program that has a data-driven execution model using the *Event-Driven Backward Chaining (EDBC)* rules developed by D. Anicic [Anicic, 2011]. Our definition of these rules however does not strictly follow their original definition but closely resembles it. The *EDBC* rules are presented here as implemented for the experimental evaluation presented in Section 6.4.1. In particular, *ELE* manipulates events which are timed-stamped ground facts, but we manipulate facts. In addition, *ELE* assigns IDs to events. When an event is retracted (i.e. a fact is deleted), its ID is used to detect and retract the dependant derived events. We do not use any ID to handle deletion.

The first step of the transformation is the binarization of rules. Each rule is transformed into a set of rules, each containing just two literals in the body. The second step is the transformation of each binary rule into a set of *add* and *delete EDBC* rules. When a fact $f$ is added or removed from the knowledge base, all relevant *add* and *delete* rules are applied by calling *add(f)* or *delete(f)* on the transformed program, respectively.

In the binarization step, each rule is assigned a unique number $k$. Then, a rule number $k$ of the form k-) $a$ :- $a_1 \wedge a_2 \wedge .. \wedge a_n$ with n literals in its body is transformed to the

following set of n binary rules where $e_{k0}$ equals *true* and $e_{kn}$ is replaced by $a$.

$$e_{k1} \text{ :- } e_{k0} \wedge a_1.$$
$$e_{k2} \text{ :- } e_{k1} \wedge a_2.$$
$$...$$
$$e_{k(n-1)} \text{ :- } e_{k(n-2)} \wedge a_{(n-1)}.$$
$$e_{kn} \text{ :- } e_{k(n-1)} \wedge a_n.$$

For an example, Listing 6.3 presents the binary rules corresponding to the first two clauses of Program 1. Please note that the head of each $i^{th}$ binary rule, corresponding to the $i^{th}$ literal in the body of rule $k$ is assigned the unique identifier $\langle k, i \rangle$ and the set of all variables appearing in the body of a binary rule is also appearing in its head.

```
1        --Binary rules of 1) a(X,Z) :- b(X,Y), c(Y,Z), d(Z)--
2 e11(X,Y) :- true ∧ b(X,Y).
3 e12(X,Y,Z) :- e11(X,Y) ∧ c(Y,Z).
4 a(X,Z) :- e12(X,Y,Z) ∧ d(Z).
5
6        --Binary rules of  2) c(Y,Z) :- e(Y,Z).--
7 c(X,Z) :- true ∧ e(Y,Z).
```

Listing 6.3: Binary rules of the program 1

The format we presented above is to give a precise form to generate binary rules. In practice, *true* clauses from conjunctions in the bodies of binary rules can be removed and when a rule has more than one binary rule, its first and second binary rules can be merged. Listing 6.4 presents the binary rules of Program 1 after applying these simplifications.

```
1 e12(X,Y,Z) :- b(X,Y) ∧ c(Y,Z).
2 a(X,Z) :- e12(X,Y,Z) ∧ d(Z).
3
4 c(X,Z) :- e(Y,Z).
```

Listing 6.4: Binary rules of the program 1

In the second step of the transformation, each binary rule $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$ is transformed to a set of *add EDBC* rules implementing the four procedures presented in Algorithm 1. These procedures work as follows. When an instance of $a_i$ is added to *ELE* (i.e. *add($a_i$)* is called on the transformed program), the first procedure encodes

this information by asserting the fact *goal($e_{k(i-1)}$, $a_i$, $e_{ki}$)* into the transformed program which is read as follows. We have an instance of $a_i$ (the second argument) and we are waiting for an instance of $e_{k(i-1)}$ (the first argument) to derive the corresponding instance of $e_{ki}$ (the third argument). When an instance of $e_{k(i-1)}$ is derived (i.e. *add($e_{k(i-1)}$)* is called), the third procedure encodes this information by asserting the fact *goal($a_i$, $e_{k(i-1)}$, $e_{ki}$)* which is read as follows. We have an instance of $e_{k(i-1)}$ (the second argument) and we are waiting for an instance of $a_i$ (the first argument) to derive the corresponding instance of $e_{ki}$ (the third argument).

When *add($a_i$)* is called, in addition to the first procedure, the second procedure is applied checking whether there is any fact of the form *goal($a_i$, $e_{k(i-1)}$, $e_{ki}$)* in the transformed program to derive the corresponding instance of $e_{ki}$. When *add($e_{k(i-1)}$)* is called, in addition to the third procedure, the fourth procedure is applied checking if there is any fact of the form *goal($e_{k(i-1)}$, $a_i$, $e_{ki}$)* in the transformed program to derive the corresponding instance of $e_{ki}$. In both cases, *add($e_{ki}$)* is called, matching *add EDBC* rules of the next binary rule and so on. The overall effect is that when a fact is added to the knowledge base, a chain of relevant add *EDBC* rules are applied, generating all facts that can be derived from the newly added fact in combination with the previous input facts and the initial set of rules.

**Algorithm 1** *EDBC add* procedures of $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$

**add**($a_i$):
assert( goal($e_{k(i-1)}$, $a_i$, $e_{ki}$) ).

**add**($a_i$):
**for all** goal($a_i$, $e_{k(i-1)}$, $e_{ki}$) **do**
    add($e_{ki}$).
**end for**

**add**($e_{k(i-1)}$):
assert( goal($a_i$, $e_{k(i-1)}$, $e_{ki}$) ).

**add**($e_{k(i-1)}$):
**for all** goal($e_{k,(i-1)}$, $a_i$, $e_{ki}$) **do**
    add($e_{ki}$).
**end for**

In the second step of transformation, a set of *delete EDBC* rules are also generated for each binary rule, implementing the four procedures presented in Algorithm 2. These procedures encode the meta-predicate *delete*, which should be called to delete a fact from *ELE*. These procedures work as follows. When an instant of $a_i$ is deleted (i.e. *delete($a_i$)* is called on the transformed program), the first procedure retracts the goal that encodes "$a_i$ has been added to the knowledge base and we are waiting for $e_{k(i-1)}$ to derive $e_{ki}$"

from the transformed program. The second procedure is also applied calling *delete($e_{ki}$)* for all $e_{ki}$ facts that were derived using $a_i$. Such calls match the delete procedures of the next binary rule and so on, propagating the deletion to dependant derived facts. The overall effect is that when a fact is deleted, a chain of delete *EDBC* rules are applied, deleting all intermediate facts and derived facts that were derived using the deleted fact. The last two procedure are similar to the first two procedures, but are when *delete* is called on instances of $e_{k(i-1)}$.

**Algorithm 2** *EDBC delete* procedures of $e_{ki}$ :- $e_{k(i-1)}$ $\wedge$ $a_i$

**delete**($a_i$):
retract( goal($e_{k(i-1)}$, $a_i$, $e_{ki}$) ).

**delete**($a_i$):
**for all** goal($a_i$, $e_{k(i-1)}$, $e_{ki}$) **do**
    delete($e_{ki}$).
**end for**

**delete**($e_{k(i-1)}$):
    retract( goal($a_i$, $e_{k(i-1)}$, $e_{ki}$) ).

**delete**($e_{k(i-1)}$):
**for all** goal($e_{k(i-1)}$, $a_i$, $e_{ki}$) **do**
    delete($e_{ki}$).
**end for**

For example, Listing 6.5 shows some of the *delete EDBC* rules of the transformed program of Program 1. The last rule in this listing is added due to subscription of the component 'comp1' to *ELE* for the facts of the form *a(X,Z)*. Suppose that we continue the example described in Section 6.3 with deleting the fact *d(3)* from *ELE* by calling *delete(d(3))* on the transformed program. Consequently, the first rule is applied and the fact *goal($e_{12}$(X,Y,3),d(3),a(X,3))* is retracted from the transformed program. The fact is removed because it says "we have *d(3)* and we are waiting for an instance of *$e_{12}$(X,Y,3)* to derive an instance of *a(X,3)*", but *d(3)* has been deleted. The second rule is also applied with the substitution *Z=3*. Applying this rule, the *goal(d(3),(*$e_{12}$*)(X,Y,3),a(X,3)* of *forall* matches the fact *goal(d(3),$e_{12}$(1,2,3),a(1,3)* in the transformed program, binding $X$ and $Y$ respectively to 1 and 2 and hence calling *delete(a(1,3))*. The call matches the last rule and the fact *a(1,3)* is sent to 'comp1' to be deleted from the set of facts matching the subscription.

```
1 delete(d(Z)): retract( goal(e12(X,Y,Z),d(Z),a(X,Z)) ).

2 delete(d(Z)): forall( goal(d(Z),e12(X,Y,Z),a(X,Z)),
                        delete(a(X,Z))).

3 delete(a(X,Z)): notify('comp1','delete',a(X,Z)).
```

LISTING 6.5: Some of the *delete* rules of the transformed program of Program 1

## 6.4 Active Queries: Naive Approach

In the previous section, we saw how a definite logic program, excluding facts, can be transformed to a set of *EDBC* rules incrementally deriving all facts that can be inferred from the program when the program is updated. While such an execution model suits on-flow processing tasks described in Chapter 3, it does not suit on-demand processing tasks for two reasons. First, explicitly representing all facts that can be inferred from the knowledge base may incur a large space complexity exhausting the memory. Second, deriving all facts that can be inferred from the program and keeping the set up-to-date after each update of the knowledge base may be computationally expensive and is a waste of resources when not all such facts are of interest for the current operational context of the robot.

We only need to derive and update the results of the active queries, queries which are of interest at the time. In this section, we present an approach based on *EDBC* rules to implement active queries. We consider a knowledge base $\langle F, P\rangle$ partitioned into two logic programs $F$ and $P$ where $F$ is a set of facts and $P$ is a set of rules containing no fact. The knowledge base is updated by asserting or retracting facts from $F$. An active query is a typical logic program query $Q$ with a unique id $Id$, represented as $\langle Q, Id\rangle$. Active queries are registered to the knowledge base with their unique identifiers, the results of which are derived and maintained up-to-date as the knowledge base is updated, until they are unregistered.

First, we consider the special case of $F$ being empty, when an active query $\langle Q, Id\rangle$ is registered to the knowledge base $\langle F, P\rangle$. In this case, the Naive approach works as follows. First, it determines the program $P'$ which is the subset of $P$ that is relevant to the query $Q$, defined below as the relevant rule set of $Q$. Then, $P'$ is transformed to the program $P''$ that has a data-driven execution model (i.e. *EDBC* rules). When the knowledge base $\langle F, P\rangle$ is updated by assertion or retraction of a fact $f$ from $F$, the

Naive approach calls *add(f)* or *delete(f)* on $P''$, respectively, incrementally updating the results of $Q$.

**Definition 30 (Relevant Rule Set).** The relevant rule set of a query $Q$ for a program $P$ is the program $P'$ that is a minimum subset of $P$ such that the results of $Q$ for both knowledge bases $\langle T, P\rangle$ and $\langle T, P'\rangle$ for an arbitrary set of facts $T$ are the same.

The relevant rule set of $Q$ is the largest set of rules from $P$ that *SLD* resolution could possibly backtrack on, when evaluating $Q$ on $P$. This set can be determined by gathering all rules in $P$ using which the query $Q$ could be possibly proven. For example, consider the first two classes of Program 1 to be the program $P$. Then, the relevant rule set of the query *a(X,Z)* contains both clauses of $P$ and the relevant rule set of the query *c(Y,Z)* contains only the second clause of $P$.

**Definition 31 (Event-Driven Backward Chaining Rule Set of Rule).** The *EDBC* rule set of a rule $R$ is the set of all *add* and *delete* *EDBC* rules generated from $R$ by applying the transformation described in Section 6.3.1.

**Definition 32 (Event-Driven Backward Chaining Rule Set of Query).** The *EDBC* rule set of a query $Q$, denoted by EDBC(Q), is the set of all rules in *EDBC* rule sets of all relevant rules of $Q$.

In addition to the *EDBC* rule set of a query $Q$, the transformed program $P''$ of $Q$ also includes the following rules to notify the *asker* component when the results of $Q$ are updated. The asker is the component who has registered $Q$. The effect of these rules is that when a new answer for $Q$ is derived, or a derived answer is deleted, the asker is informed thorough calling the corresponding notify function.

*add(Q) :- notify(Id, 'add', Q).*
*delete(Q) :- notify(Id, 'delete', Q).*

The *EDBC* rule set of a query generates answers of the query if the set $F$ of facts in the knowledge base $\langle F, P\rangle$ is initially empty. An active query is however registered to the knowledge base at runtime where some facts may exist in $F$. Therefore, when a query is registered, we determine the set of its *EDBC* rules and call *add(f)* for every fact $f$ in $F$ that is relevant to $Q$. We call the latter step *initialization* by which we generate all

*goal* facts and results of the query that can be derived from the facts already exist in $F$. After this, when $F$ is updated by assertion or retraction of facts, the corresponding *add* and *delete* meta-predicates are called on the transformed program, keeping the results of the query up-to-date. The relevant facts of a query are the facts whose types are included in the relevant predicate set of the query defined below.

**Definition 33 (Relevant Predicate Set).** The relevant predicate set of a query $Q$ for a program $P$ is the largest set of predicate types on the term of which the *SLD* method could possibly backtrack, when evaluating $Q$ on $P$.

The relevant predicate set of $Q$ can be determined by gathering all predicate types appearing in the relevant rule set of $Q$. For example, the relevant predicate set of the query *a(X,Z)* for program 1 contains the predicate types *b/2, c/2, d/1, e/2*.

The *EDBC* rule set of a query $Q ::= p_n(T_1, \ldots, T_n)$[4] generates all facts of type $p_n$. However, some variables in arguments $T_i$ of the query may be bound to some variables. This information can be taken into account to respectively bind the variables in *EDBC* rule set of the query to generate the only facts that are answers to the query according to the given variable substitution. Binding variables in *EDBC* rules results in their activation by less number of facts, only the ones which are relevant to the query, and hence the computational performance increases.

**Definition 34 (Substituted *EDBC* Rule Set of Query).** The substituted *EDBC* rule set of a query $Q$, denoted by $EDBC_s(Q)$, is generated by applying the maximal substitution on *EDBC* rule set of $Q$, such that those rules generate a fact if and only if such fact is a result for $Q$ (i.e. can be unified with $Q$).

The following summarises the Naive approach for implementing active queries on a knowledge base being a union of a set of facts $F$ and a set of rules $P$. At the initialization phase of the system, create an empty *Prolog* program $P''$. When a query is registered, first determine its substituted *EDBC* rules for the program $P$ and add them to $P''$. Then, for each fact $f$ in $F$ which belongs to the relevant predicate set of $Q$, call *add(f)* on $P''$. When the knowledge base is updated by asserting or retracting a fact $f$ from $F$, respectively call *add(f)* or *delete(f)* on $P''$. When the query is unregistered, remove its substituted *EDBC* rules and the goal facts generated by these rules from $P''$.

[4] If the query is a conjunction of terms $Q_1 \wedge \ldots Q_n$, we add a corresponding rule $Q :- Q_1 \wedge \ldots Q_n$. to the knowledge base and replace the query with $Q$. $Q$ is given a predicate name not conflicting with existing predicate symbols in the belief base.

The substituted *EDBC* rule sets of two queries may have some rules in common which should be taken into account in implementation of the Naive approach. To avoid side-effects of the *EDBC* rules of different queries on each other, we encode the $Id$ of each query in the *EDBC* rules added to $P''$ for that query. In this way, the *goal* and derived facts generated by *EDBC* rules of a query are made local to the query.

### 6.4.1 Evaluation

The only robotic system supporting a form of active query mechanism is *ORO*. However, *ORO* does not present any detail on the performance of its active query mechanism. *Pellet*, the reasoning engine of *ORO*, does not support incremental query evaluation in general, for instance, when the knowledge base contains rules. Therefore, it is not possible to compare our approach with *ORO*. Instead, we evaluate our approach by demonstrating its use to evaluate logical queries in the 2APL agent programming language and discussing its performance. In the next chapter, we consider the plan execution requirements of agent programming languages in robotic applications and discuss the benefits of using active queries in planning and plan execution monitoring to address the requirements.

2APL is a well-known *BDI*-based agent programming language developed and maintained at Utrecht University. In the 2APL program of an agent, there is a set of *pg-rules* used to generate plans to achieve the goals of agent. The execution of the agent program is cyclic and in each cycle, all pg-rules are evaluated, each may include a query to be evaluated on the agent's belief base. To deal with performance issues caused by the repeated evaluation of these belief queries, 2APL has recently implemented the caching mechanism described in the introduction of this chapter. In this section, we first describe how *pg-rules* are evaluated in the 2APL deliberation cycle, implementing the aforementioned caching mechanism. We then describe how the Naive approach presented for implementing active queries, can be used to evaluate belief queries of pg-rules, implementing them as active queries. Finally, we compare the performance of our approach with that of the caching mechanism currently used by 2APL on an agent program.

#### 6.4.1.1 Belief Query Evaluation in 2APL

A 2APL program of an agent has a set of pg-rules of the form $[\langle goalquery \rangle]$" $\leftarrow$ "$\langle belquery \rangle$"|"$\langle \pi \rangle$ where *goalquery* is a query to the goal base (i.e. $GB$) of the agent, *belquery* is a query to the belief base (i.e. $BB$) of the agent and $\pi$ is a plan template containing variables.

A pg-rule is applied when the goal query can be derived from the agent's goals and the belief query can be derived from the agent's beliefs. These queries result in substitution of variables by applying which the plan is instantiated and added to the plan base (i.e. $PB$) to be executed by the agent. The agent goal base consists of an independent list of goals. Therefore the *goalquery* can be derived from different agent's goals resulting in different sets of substitutions. This means a pg-rule can be applied more than once for different agent's goals. The belief query can also have more than one result. 2APL does not generate a separate plan for each of such results but only for one of them. This is realized by checking whether there is currently no plan being executed which has been created by applying the same pg-rule for the same goal.

Algorithm 3 presents a part of the 2APL delibration cycle in which the pg-rules are applied, which is read as follows. For each pg-rule of which the head is true, if the rule has not been applied to generate a plan currently being executed and the belief query results in a substitution $\theta$, then $\theta$ is applied on the plan $\pi$ and the instantiated plan is added to the plan base to be executed.

For each pg-rule of which the head is a goal query, for each distinguished substitution $\beta$ using which the goal query can be derived from the agent goal base, do the following. If the goal has not been achieved (i.e. it is not inferred from the belief base) and $\beta$ has not been used for this rule to generate a plan being executed, then $\beta$ is applied on the belief query, and if the instantiated belief query can be derived from the agent belief base using a substitution $\theta$, then the substitutions $\beta$ and $\theta$ are applied to instantiate the plan and the plan is added to the plan base. In this algorithm, queries are posted to the belief base only if there is no cache available for them or if the cache has been invalidated. The cached query results are used otherwise.

**Algorithm 3** Evaluation of pg-rules in 2APL Deliberation Cycle

```
for all pg-rule R::=goalQ ← belQ|π in agent program do
    if goalQ is true then
        if π ∉ PB ∧ BB⊢θ belQ then
            Add πθ to PB
        end if
    else
        for all β that GB⊢β goalQ do
            if BB ⊬ goalQβ ∧ πβ ∉ PB ∧ BB⊢θ belQβ then
                Add πβθ to PB
            end if
        end for
    end if
end for
```

#### 6.4.1.2 Naive Approach for Data-Driven Belief Query Evaluation in 2APL

Algorithms 4 and 5 present an approach for implementing the belief queries of pg-rules as active queries using the Naive approach. Algorithm 4 presents the procedures that are performed at compile time which are the followings. We generate relevant rule sets, relevant predicate sets and *EDBC* rule sets of belief queries of pg-rules. In addition, for each pg-rule of which the head is true, we generate and activate (i.e. add to the belief base and initialize) their substituted EDBC rules because as soon as the belief query of such a rule has a result, the rule can be applied if a corresponding plan is not being executed. Therefore we are always interested on results of such belief queries.

**Algorithm 4** Naive Approach for Data-Driven Evaluation of pg-rules in 2APL Deliberation Cycle - Compile Time

**for all** Prolog rule $R$ in agent program belief base **do**
  generate *EDBC-Rule-Set(R)*
**end for**
**for all** pg-rule $R$::=$goalQ \leftarrow belQ|\pi$ in agent program **do**
  generate *Relevant-Rule-Set(belQ)*
  generate *Relevant-Predicate-set(belQ)*
  generate *EDBC(belQ)*
  **if** *goalQ* is *true* **then**
    generate $EDBC_s$*(belQ)*
    add $EDBC_s$*(belQ)* to *BB*
    **for all** facts $p_n(t_1, \ldots, t_n) \in BB \mid p_n \in$ Relevant-Predicate-set(belQ) **do**
      *call(add*$(p_n(t_1, \ldots, t_n))$ *on BB*
    **end for**
  **end if**
**end for**

At runtime, pg-rules are evaluated using the procedures presented in Algorithm 5. In summary, this algorithm works as follows. Whenever the goal query of a pg-rule can be derived from the goal base using a substitution, the substitution is applied on *EDBC rules* of the belief query of the pg-rule, generating the corresponding substituted $EDBC_s$ rules. The $EDBC_s$ rules are added to the belief base and are initialized, generating the results of the belief query and maintaining the results up-to-date as facts are added or deleted from the belief base. The results of a query are sent from the belief base to the agent *Java* program and are maintained in a *Hash Map*. As facts are added or removed from the belief base, *add* or *delete* meta-predicates of $EDBC_s$ rules are called[5] updating the query results maintained in the agent program thorough sending notifications. Whenever the belief query has a result and a plan for the given goal is not being executed, the pg-rule is applied, generating a plan. The $EDBC_s$ rules of a

[5]Adding or deleting facts occurs in another part of the agent program and therefore is not shown in Algorithm 5.

belief query are deleted whenever the agent has no longer the goal for which the *EDBC* rules were substituted. For the case of pg-rules whose head is true, their corresponding $EDBC_s$ rules are always in the belief base and hence the results of their belief queries are always maintained up-to-date.

**Algorithm 5** Naive Approach for Data-Driven Evaluation of pg-rules in 2APL Deliberation Cycle - Run Time

```
for all pg-rule R::=goalQ ← belQ|π in agent program do
    if goalQ is true then
        if π ∉ PB ∧ Results(belQ) != ∅ then
            unify(Results(belQ).Element(),belQ)_θ
            Add π_θ to PB
        end if
    else
        for all β that GB⊢_β goalQ do
            if BB ⊬ goalQ_β ∧ π_β ∉ PB then
                if not added EDBC_s(belQ_β) then
                    generate EDBC_s(belQ_β)
                    add EDBC_s(belQ_β) to BB
                    for all facts p_n(t_1,...,t_n)∈BB | p_n ∈
                        Relevant-Predicate-set(belQ) do
                        call(add(p_n(t_1,...,t_n)) in BB
                    end for
                end if
                if Result-Set(belQ_β) != ∅ then
                    unify(Results(belQ).Element(),belQ)_θ
                    Add π_βθ to PB
                end if
            end if
        end for
        for all EDBC_s(belQ_β) in BB do
            if GB ⊬ goalQ_β then
                delete EDBC_s(belQ_β) from BB
                delete results(belQβ)
            end if
        end for
    end if
end for
```

#### 6.4.1.3 Empirical Results

This section discusses the performance of the Naive approach for incremental evaluation of belief queries of pg-rules by comparing it with that of the caching mechanism currently used in 2APL. The performance of these approaches depends on many factors including the nature of the knowledge base, queries and updates of the knowledge base which are application dependent. Therefore, a fair evaluation requires the comparison

of performance on a variety of typical applications of agent programming languages. There is a number of exemplary applications, the implementation of which are available for a number of agent programming languages [Alechina, 2013], that are good candidates for the purpose of evaluation. However, the implementation of these applications often include *Prolog* built-in predicates such as *negation as failure* which are not yet supported in our approach. Consequently, we compare the performance on the following application scenario to highlight its main pros and cons.

The scenario we consider is as follows. An agent receives goals of the form *need(O)* representing the need to buy an object of type $O$. It also receives events of the form *addSeller(S,T)* representing the availability of a seller $S$ of type $T$ and events of the form *addOffer(S,O,P)* representing an offer by a seller $S$, selling an object of type $O$ with a price $P$. The agent has one pg-rule *need(O)* $\leftarrow$ *accessible(O, S)*| {*dropgoal(need(O)); @env(newgoal(true),Return);*} . This rule is read as "when the agent needs object $O$, if it believes that seller $S$ offers $O$, it drops the goal (i.e. buys it) and asks for a new goal". The agent belief base contains the rule *accessible(O, S) :- seller(S, T), has(T,O), offers(S, O, P)*. This rule is read as "an object of type $O$ is accessible from seller $S$, if the seller is of type $T$ and sellers of type $T$ sell objects of type $O$ and the seller has an offer for selling an object of type $O$".

The following usecases are evaluated on an *XPS Dell* laptop with *Intel Core i7 CPU @ 2.10GHz x 4* running 64 bits *Ubuntu 12.04 LTS*.

- Usecase 1: there are 10 types of seller, 100 unique object types per each type of seller (i.e. 1000 object types in total) and 100 sellers, each offering 10 objects. The agent is always given goals which are available in the market.
- Usecase 2: this usecase is similar to the usecase 1, but there are 50 types of seller, 100 unique object types per each type of seller (i.e. 5000 object types in total) and 100 sellers, each offering 50 objects.
- Usecase 3: this usecase is similar to the usecase 1, however agent is given a goal which is never available in the market. At each deliberation cycle, the market is updated by adding one new seller offering one object.
- Usecase 4: this usecase is similar to the usecase 2, however agent is given a goal which is never available in the market. At each deliberation cycle, the market is updated by adding one new seller offering one object.
- Usecase 5: this usecase is similar to the usecase 3, however the market is updated by adding 10 new offers and deleting 10 old offers in each deliberation cycle.

| Multi Cycle Caching | Cycles | Averag-time |
|---|---|---|
| Usecase 1 | 415560 | 1,442 |
| Usecase 2 | 92070 | 6,515 |
| Usecase 3 | 12145 | 49,391 |
| Usecase 4 | 5095 | 117,718 |
| Usecase 5 | 2077 | 286,377 |
| Usecase 6 | 1002 | 592,248 |

TABLE 6.1: Performance of the caching mechanism

- Usecase 6: this usecase is similar to the usecase 5, however the market is updated by adding 30 new offers and deleting 30 old offers in each deliberation cycle.

We evaluate the performance by running each usecase once using the current version of 2APL that uses caching and once using the modified version of 2APL that uses the naive approach. We run each usecase for 10 minutes and measure the number of deliberation cycles performed and the average deliberation cycle time in milliseconds. As belief queries are performed faster, the average deliberation cycle is less and the program performs more deliberation cycle. In our scenario, performing more deliberation cycle means the agent achieves more goals in its 10 minutes run.

Tables 6.1 and 6.2 present the performance of caching and Naive approaches, respectively. In usecases 1 and 2, each belief query succeeds at its first attempt. The performance results for these usecases show that in such case, the average deliberation cycle time of the caching approach is 2 to 3 times less. Taking into account that the agent execution contains other steps such as processing events and executing plans, this means caching approach performs more than 2 to 3 times better when the first evaluation of belief queries is succeeded. The results are as expected, because the Naive approach pays extra prices to add and remove *EDBC* rules and *goal* facts and it does not bring any advantage as no query is incrementally evaluated.

In usecases 3 and 4, one goal is given to the agent and the corresponding belief query is repeatedly executed in every deliberation cycle. The results show that if queries are repeated while the knowledge base goes through small changes, the cost of data-driven approach is extremely low, tens of times less than the cost of caching mechanism. Finally, usecases 5 and 6 show that as larger as the number of updates happening in between the repetition of a query, the more is the cost of the data-driven approach until a point that the caching mechanism performs better.

| Data-Driven Queries | Cycles | Averag-time |
|---|---|---|
| Usecase 1 | 143550 | 4,174 |
| Usecase 2 | 38640 | 15,522 |
| Usecase 3 | 211077 | 2,841 |
| Usecase 4 | 207382 | 2,891 |
| Usecase 5 | 2062 | 290,779 |
| Usecase 6 | 967 | 614,847 |

TABLE 6.2: Performance of the Naive active query mechanism

## 6.5 Active Queries: Optimized Approach

The Naive approach suffers from inefficiency in performing the following tasks that are addressed by the Optimized approach developed in this section:

1. To register an active query, a set of corresponding *EDBC* rules are generated.
2. The relevant substitution is computed and applied to the *EDBC* rules.
3. Substituted $EDBC_s$ rules are added to the knowledge base.
4. The procedure *add(f)* is called for all relevant facts $f$ in the knowledge base.
5. The *EDBC* rules of the query and intermediate results are deleted when the query is unregistered.

When the set of queries to be performed in the life time of the program is known in advance, as is the case in agent programming languages, then the first task can be performed at compile time. The rest are however to be performed at runtime. Among these tasks, the most expensive one in terms of both computation and memory is the fourth task. Examining the computations performed by *EDBC* rules of a *Horn* clause, one can notice that every fact that matches a literal in the body of the clause is remembered by *EDBC* rules by asserting a corresponding *goal* fact. Consequently, not only does the fourth task creates multiple copies of the facts in the knowledge base, but also it can be very expensive as all relevant facts are read from the knowledge base and copies of which are asserted as *goal* facts. These copies are then retracted when the query is unregistered. Read and write operations are of the most expensive in *Prolog* and thus is an important subject of optimization.

This section describes the Optimized approach for source code transformation of definite logic programs into *active logic programs* that support active queries. Given a logic program $P_o$ (i.e original program), the Optimized approach transforms it to a *Prolog* program $P_t$ (i.e. transformed program) such that the results of a query $q$ for both $P_o$

and $P_t$ are the same. The difference between $P_o$ and $P_t$ is when they are updated by addition or deletion of facts. When $P_o$, executed by a *Prolog* system, is updated after the evaluation of a query, the query should be re-evaluated from scratch to update the results of the query. In contrary, queries can be (un-)registered to $P_t$ as active queries. When $P_t$ is updated, it efficiently updates the results of the active queries in an incremental manner. To update $P_t$ by addition or deletion of a fact $f$, we assert $f$ in $P_t$ or retract $f$ from $P_t$ respectively using *assert(f)* or *retract(f)*, as in a normal *Prolog* program. In addition, we respectively call *add(f)* or *delete(f)* on $P_t$ to update the results of active queries.

A query to a transformed logic program is of the form $\langle \mathrm{q}, \mathrm{id} \rangle$ where $q$ is a valid query to the original logic program and *id* is the unique id of the query. When a query is registered to a *Prolog* system executing the transformed program, the system answers with results of the query and notifies about the changes of the query results when the program is updated by addition or deletion of facts. At the system architecture level, the *Prolog* execution system is provided with a wrapper to enable the execution of callback functions whenever query results are updated. Query result updates are sent to components who have registered the queries.

The following describes how the Optimized approach transforms a definite logic program $P_o$ into an active logic program $P_t$, in a few steps. At first, we present a transformation algorithm where the transformed program is able to derive some answers of queries. Then, we gradually complete the algorithm such that the transformed program derives all answers of queries.

Given an original program $P_o$, $P_t$ is constructed as follows. First, we add all clauses of the form *a :- true* (i.e. facts) from $P_o$ to $P_t$. Then, for each rule in $P_o$ with $n$ literals in the body where $n \geq 1$, $n+1$ *Optimized approach Active Query (OAQ)* rules are generated and added to $P_t$ as follows. Each rule $a$ :- $a_1, a_2, \ldots a_n$ is assigned a unique number $k$, from which the following $n$ intermediate binary rules are generated. The term $e_{k_0}$ in the first binary rule represents a term constructed by the "$e_{k0}$" functor symbol and the arguments of the term $a$.[6]

$$e_{k1} \text{ :- } e_{k0} \wedge a_1.$$

$$e_{k2} \text{ :- } e_{k1} \wedge a_2.$$

...

$$e_{kn} \text{ :- } e_{k(n-1)} \wedge a_n.$$

[6]Note that in the binarization step described in Section 6.3.1, $e_{k0}$ was replaced by *true*.

For example, the first clause of Program 2 presented in Listing 6.6 is converted to the binary rules presented in Listing 6.7. In generating the binary rules, the original clause has been given the number 1. As shown in the example, each term $e_{ki}$ in the head of a binary rule has the terms $e_{k0}, a_1, a_2, \ldots a_i$ as its arguments, the reason for which will be explained in Section 6.6.3.

```
1 a(X) :- b(X,Y), c(Y,Z).
2 b(0,1). b(1,2). c(2,3).
```

Listing 6.6: Program 2

```
1 e11(e10(X),b(X,Y)) :- e10(X), b(X,Y).
2 e12(e10(X),b(X,Y), c(Y,Z)) :- e11(e10(X),b(X,Y)), c(Y,Z).
```

Listing 6.7: Binary rules of Program 2

So far, we have transformed each rule $k$ of $P_o$ that has $n$ literals in the body into $n$ binary rules. Now, for each rule $k$, the rule presented in Algorithm 6 is added to $P_t$. Also, for each binary rule $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$ of a rule $k$ $(1 \leq i \leq n)$, the rule presented in Algorithm 7 is added to $P_t$.[7] Therefore in total, each rule in $P_o$ is transformed into *n+1* *OAQ* rules in $P_t$. In these algorithms, $e_{k(i-1)/\theta_i}$ represents the term derived by applying the substitution $\theta_i$ to the term $e_{k(i-1)}$ and $P_t \models_{\theta_j} a_{i/\theta_i}$ means that $a_{i/\theta_i}$ is derived from the program $P_t$ by applying the substitution $\theta_j$. As $P_t$ and $P_o$ only shares the facts of $P_o$, $P_t \models_{\theta_j} a_{i/\theta_i}$ means that there is a fact in the original program matching $a_{i/\theta_i}$ by applying the substitution $\theta_j$.

**Algorithm 6** For each rule $a$ :- $a_1, a_2, \ldots a_n$

**add**$(e_{kn/\theta_{n+1}})$ : add$(a_{/\theta_{n+1}})$.

**Algorithm 7** For each binary rule $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$

**add**$(e_{k(i-1)/\theta_i})$ :
**for all** $a_{i/\theta_i}$ such that $P_t \models_{\theta_j} a_{i/\theta_i}$ **do**
    add$(e_{ki/\theta_i\theta_j})$.
**end for**

For a query $q$ matching the head $a$ of a rule number $k$ in $P_o$, the corresponding *OAQ* rules in $P_t$ are used to generate the answers of $q$ that can be inferred by applying the rule $k$. A query $q$ matches $a$ if there is a substitution of variables $\theta_1$ that unifies $a$ and $q$ represented by $unify(a,q)_{\theta_1}$. To derive answers of $q$ using the rule $k$ and given $unify(q,a)_{\theta_1}$, we call $add(e_{k0/\theta_1})$ on the transformed program $P_t$. If we ignore the intermediate generated goals of types $e_{k1} \ldots e_{kn}$, calling $add(e_{k0/\theta_1})$ generates chains of computations that result in

[7] The rule is in Pseudo code.

finding some answers of $q$ that are derivable by applying the *SLD* resolution strategy on the rule $k$ in the original program $P_o$. Only some answers are found, because each sub-goal $a_{i/\theta_i}$ is resolved only using facts. In *SLD* resolution, a sub-goal can also be resolved using rules. We will later extend our algorithm accordingly, but let us first see an example of how *OAQ* rules work.

Listing 6.8 presents the transformed program $P_t$ of Program 2 from Listing 6.6. To evaluate the query *a(X)* on Program 2, we call *add(e10(X))* on $P_t$. The call matches the head of the first rule, calling *add(e11(e10(X),b(X,Y))))* for each *b(X,Y)* in $P_t$. First, *b(X,Y)* matches *b(0,1)* and *add(e11(e10(0),b(0,1))))* is called, matching the second rule, binding $X$ to 0 and $Y$ to 1. Consequently, the sub-goal *c(1,Z)* in the body of the second rule is called which fails. Then *b(X,Y)* matches *b(1,2)*, calling *add(e11(e10(1),b(1,2)))* which matches the second rule, binding $X$ to 1 and $Y$ to 2. The sub-goal *c(2,Z)* is resolved using the fact *c(2,3)*, binding $Z$ to 3. Consequently, *add(e12(e10(1),b(1,2), c(2,3)))* is called, matching the third rule, generating the answer *a(1)*. By "generating the answer *a(1)*", we mean that *add(a(1))* is called which should be captured to derive the answer *X=1* for the query *a(X)*. In the example, note how the use of variable substitutions guiding the generation of intermediate results toward generating answers of the given query. Note also that answers are derived by a depth-first search strategy as is the case for the *SLD* resolution algorithm. In this way, if we wish to find only one answer for a given query, we can stop the computation as soon as an answer is found.

```
1 add(e10(X)) :- forall(b(X,Y),
2                       add(e11(e10(X),b(X,Y))) ).
3
4 add(e11(e10(X),b(X,Y)) :- forall(c(Y,Z),
5                                   add(e12(e10(X),b(X,Y), c(Y,Z))).
6
7 add(e12(e10(X),b(X,Y), c(Y,Z))) :- add(a(X)).
8
9 b(0,1). b(1,2). c(2,3).
```

Listing 6.8: Transformed program of Program 2

Given a query $q$ and a rule number $k$ whose head $a$ is unified with $q$ by applying a substitution $\theta$, the above example shows how calling $e_{k_0/\theta}$ derives answers of $q$ that can be derived using the rule $k$ and facts. We call this activating the rule $k$ for the substitution $\theta$. A query can match the head of a number of rules. To apply all such rules to search for answers of the query, we activate all such rules with corresponding substitutions. In addition, some results of the query can be present in $P_t$ as facts imported directly from $P_o$ which should be taken into account. Putting all these together, Algorithm 8

presents the *perform* procedure to perform a query $Q$ on $P_t$. We generate such a *perform* procedure for each unique functor-symbol/arity in $P_o$ and add it to $P_t$.

**Algorithm 8** Performing a query

**perform**($q$) :
**for all** rule number $k$ in $P_o$ with the head $a_k$ such that *unify(q, $a_k$)*$_{\theta_k}$ **do**
add($e_{k0/\theta_k}$).
**end for**
**for all** facts $f_k$ in $P_t$ such that *unify(q,$f_k$)*$_{\theta_k}$ **do**
add($f_{k/\theta_k}$).
**end for**

When a rule $a$ :- $a_1, a_2, \ldots a_n$ is activated by a substitution $\theta$, the corresponding *OAQ* rules generate results of $a_{/\theta}$, searching over the facts to resolve each sub-goal $a_{i/\theta_i}$. However, a sub-goal $a_{i/\theta_i}$ could be also resolved by applying a rule number $h$ whose head matches the sub-goal by applying the substitution $\theta_h$. Therefore, we need to activate all such rules $h$ by calling $e_{h0/\theta_i\theta_h}$ to generate answers of the sub-goal $a_{i/\theta_i}$ that can be derived using those rules. Therefore, we accordingly revise Algorithm 7 into Algorithm 9. The activation of rules are goal-directed as the relevant substitutions are applied to avoid computing irrelevant answers to sub-goals.

**Algorithm 9** For each binary rule $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$

**add**($e_{k(i-1)/\theta_i}$) :
**for all** $a_{i/\theta_i}$ such that $P_t \models_{\theta_j} a_{i/\theta_i}$ **do**
add($e_{k_i/\theta_i\theta_j}$).
**end for**
**for all** rule number $h$ in $P_o$ with the head $a_h$ such that *unify*($a_{i/\theta_i}, a_h$)$_{\theta_h}$ **do**
activate the rule number $h$ by the substitution $\theta_i\theta_h$
(i.e. add($e_{h0/\theta_i\theta_h}$) )
**end for**

For a sub-goal $a_{i/\theta_i}$, Algorithm 7 activates each rule $h$ that $a_h$, the head of the rule $h$, matches the sub-goal by applying a substitution $\theta_h$. When such a rule $h$ is activated, the *OAQ* rules of $h$ generate answers of the sub-goal that can be derived by applying the rule $h$. When such an answer $a_{i/\theta_i\theta_h\theta_l}$ is derived, *add($a_{i/\theta_i\theta_h\theta_l}$)* is called. However, Algorithm 7 does not capture the call to resolve the sub-goal using the derived answer. To resolve the sub-goal using the answers derived by the activated rules, we revise Algorithm 9 into Algorithm 10, which works as follows. When *add($e_{k(i-1)/\theta_i}$)* is called, it asserts *goal($a_{i/\theta_i}$, $e_{k_i/\theta_i}$)* into $P_t$ which is read as "we are waiting for a fact that matches $a_{i/\theta_i}$ to generate $e_{k_i/\theta_i}$ (i.e. to call *add($e_{k_i/\theta_i}$)* )." When a fact $a_{i/\theta_i\theta_h\theta_l}$ is derived by an activated rule and hence add($a_{i/\theta_i\theta_h\theta_l}$) is called. The call matches the last procedure, calling *add($e_{k_i/\theta_i\theta_h\theta_l}$)* for each recorded *goal*($a_{i/\theta_i}$, $e_{k_i/\theta_i}$). In this way, the answers derived by activated rules are used to resolve the sub-goals.

**Algorithm 10** For each binary rule $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$

**add**$(e_{k(i-1)/\theta_i})$ :
**for all** $a_{i/\theta_i}$ such that $P_t \models_{\theta_j} a_{i/\theta_j}$ **do**
    add$(e_{ki/\theta_1\theta_j})$.
**end for**
assert(goal$(a_{i/\theta_i}, e_{k_i/\theta_i})$).
**for all** rule number $h$ in $P_o$ with the head $a_h$ such that $unify(a_{i/\theta_i}, a_h)_{\theta_h}$ **do**
    activate the rule number $h$ by substitution $\theta_i\theta_h$
    (i.e. add$(e_{h_0/\theta_i\theta_h})$ )
**end for**

**add**$(a_{i/\theta_i\theta_h\theta_l})$ :
**for all** goal$(a_{i/\theta_i}, e_{ki/\theta_i})$ **do**
    **add**$(e_{ki/\theta_i\theta_h\theta_l})$
**end for**

For example, Listing 6.10 presents the transformed program of Program 3 from Listing 6.9.[8] To evaluate the query *a(0,Z)* on Program 3, we call *add(e10(0,Z))* on the transformed program, because the query matches the head of the first rule of Program 3 using the substitution *X=0*. The call matches the first rule in $P_t$, binding $X$ to 0. Consequently, for each *b(0,Y)* in $P_t$, *add(e11(e10(0,Z),b(0,Y)))* is called. The sub-goal *b(0,Y)* is resolved using the fact *b(0,1)* and *add(e11(e10(0,Z),b(0,1)))* is called. The call matches the second and third rules, binding $X$ to 0 and $Y$ to 1. The second rule has no effect because *c(1,Z)* does not match any fact. The third rule asserts *goal(c(1,Z),e12(e10(0,Z),b(0,1),c(1,Z)))* into $P_t$ and then calls *add(e20(1,Z))*. The effect is that the second rule of $P_o$ is activated to derive answers for the sub-goal *c(1,Z)* because its head *c(X,Y)* matches the sub-goal by binding $X$ to 1 and renaming $Y$ to $Z$. The call *add(e20(1,Z))* matches the fifth rule by binding $X$ to 1 and renaming $Y$ to $Z$. The sub-goal *d(1,Z)* in the body is resolved using *d(1,0)*, calling *add(e21(e20(1,0),d(1,0)))* which matches the sixth rule, calling *add(c(1,0))*. The call matches the seventh rule, binding $Y$ to 1 and $Z$ to 0. The seventh rule uses the answer *c(1,0)* that has been derived for the sub-goal *c(1,Z)*, as follows. When the body of the seventh rule is evaluated, *goal(c(1,0),e12(e10(X,0),b(X,1), c(1,0)))* is matched with *goal(c(1,Z),e12(e10(0,Z),b(0,1),c(1,Z)))* which was previously asserted, calling *add(e12(e10(0,0),b(0,1),c(1,0)))*. The call matches the fourth rule, deriving the answer *a(0,0)* for the query *a(0,Z)*.

```
1 a(X) :- b(X,Y), c(Y,Z).
2 c(X,Y):- d(X,Y).
3 b(0,1). b(1,2). c(2,3). d(1,0). d(5,3).
```

[8] The *perform* procedures are not presented for brevity.

Listing 6.9: Program 3

```
1 add(e10(X,Z)):- forall(b(X,Y),
                          add(e11(e10(X,Z),b(X,Y))) ).
2 add(e11(e10(X,Z),b(X,Y)):- forall(c(Y,Z),
                                  add(e12(e10(X,Z),b(X,Y), c(Y,Z))).
3 add(e11(e10(X,Z),b(X,Y)):-
                assert(goal(c(Y,Z),e12(e10(X,Z),b(X,Y), c(Y,Z)))),
                add(e20(Y,Z)).
4 add(e12(e10(X,Z),b(X,Y), c(Y,Z))):- add(a(X,Z)).
5 add(e20(X,Y)):- forall(d(X,Y),
                           add(e21(e20(X,Y),d(X,Y)))).
6 add(e21(e20(X,Y),d(X,Y))):- add(c(X,Y)).
7 add(c(Y,Z)):- forall(goal(c(Y,Z),e12(e10(X,Z),b(X,Y), c(Y,Z))),
                          add(e12(e10(X,Z),b(X,Y), c(Y,Z)).
8 b(0,1). b(1,2). c(2,3). d(1,0).
```

Listing 6.10: Transformed program of Program 3

The evaluation of different queries may include the evaluation of sub-goals of the same type. Even the evaluation of one query may include the evaluation of sub-goals of the same type. When sub-goals of the same type are evaluated, they may be the same sub-goals or different ones due to different substitutions. Evaluations of sub-goals activate those rules whose head match the sub-goals. In our algorithm, corresponding rules are activated by the number of times the sub-goals are evaluated. Consequently, the same rule may derive the same answer a number of times due to its activation by a number of sub-goals. To make the derived answers local to each sub-goal, we include an $Id$ argument in $e_{ki}$ terms of the binary rules. In this way, each rule is activated to derive answers of a specific query/sub-goal and derived answers are consumed by the corresponding query/sub-goal. For each active query $\langle q, id \rangle$, the $id$ is used to activate the rules whose heads match the query and is passed to all other rules, activated to derive answers of the sub-goals of the query. When a query is unregistered, its $id$ is used to efficiently deactivate the query by deleting all *goal* facts (i.e. intermediate results) generated for the query. Activating a rule multiple times for the same sub-goals is a redundant task. We will further develop the Optimized approach into the Tabled Optimized approach that allows re-using the sub-goal results whenever possible.

### 6.5.1 Incremental Update

Algorithm 10 not only does enable us to utilize the answers generated for sub-goals by activating their corresponding rules, but also inherently supports an incremental update of query results when the knowledge base is updated by adding or removing facts. To add a fact $f$ to the knowledge base (i.e. logic program), we perform two operations. First, we assert the fact to the knowledge base. This step is required because, as observed in Algorithm 10, facts are used to resolve sub-goals when an active query is registered and evaluated for the first time. Second, we perform $add(f)$ to update the results of the active queries.

To explain how $add(f)$ updates the query results, consider a binary rule of the form $e_{ki}$:-$e_{k(i-1)} \wedge a_i$. When active queries are evaluated for the first time, some *goals* of the form goal($a_{i/\theta_i}$, $e_{k_i/\theta_i}$) are asserted in the knowledge base. When a fact $f_j$ is added to the knowledge base and $add(f_j)$ is called, for each such a *goal* such that $unify(f_j, a_{i/\theta_i})_{\theta_j}$, $add(e_{k_i/\theta_i\theta_j})$ is called, as if $f_j$ was derived by an activated rule for the sub-goal $a_{i/\theta_i}$. For example, consider updating Program 3 by adding the fact *c(1,3)* where the query *a(0,Z)* is active. First, *c(1,3)* is asserted in $P_t$ and then *add(c(1,3))* is called. The call matches the seventh rule, binding $Y$ to 1 and $Z$ to 3. Then, *goal(c(1,3),e12(e10(X,3),b(X,1), c(1,3)))* is matched with *goal(c(1,Z),e12(e10(0,Z),b(0,1),c(1,Z)))* which was asserted when evaluating the query *a(0,Z)*, calling *add(e12(e10(0,3),b(0,1),c(1,3)))*. The call matches the fourth rule, deriving the answer *a(0,3)* for the query *a(0,Z)*.

The way the *OAQ* rules update query results is similar to the way the *add EDBC* rules update the query results. The difference is that the *add EDBC* rules of a binary rule of the form $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$ generate two types of *goal* facts but the *OAQ* rules generate one type. In *ELE*, *goal* facts of the forms *goal*($e_{k(i-1)}$, $a_i$, $e_{ki}$) and *goal*($a_i$, $e_{k(i-1)}$, $e_{ki}$) are generated. The optimized approach however generates only *goal* facts of the form *goal*($a_i$, $e_{ki}$). Consequently, the Optimized approach directly stores the facts in the knowledge base and asserts *goal* facts only for derived intermediate results. As opposed, *ELE* creates a separate copy of relevant facts for each binary rule.

To handle incremental deletion of facts, the transformed program includes also the set of *OAQ delete* rules presented in Algorithm 11 for each binary rule. When a fact $f$ is deleted, we retract it from $P_t$ and then call *delete(f)*, propagating the deletion to accordingly update the query results. The definition of *OAQ delete* rules is also similar to the definition of *delete EDBC* rules. The difference is that as the Optimized approach directly stores the facts, a fact can be simply removed by retracting it from the knowledge base, rather than the need for removing multiple copies of it stored as *goal*($e_{k,i-1}$, $a_i$,

$e_{k,i}$*)* facts by *add EDBC* rules. After retracting the fact, retraction is propagated by *delete OAQ* rules as in *ELE*.

**Algorithm 11** *OAQ delete* rules of each $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$

**delete**($a_{i/\theta_i}$):
**for all** goal($a_{i/\theta_i}$, $e_{k_i/\theta_i}$) **do**
    delete($e_{ki/\theta_i}$).
**end for**

**delete**($e_{k(i-1)/\theta_i}$):
retract(goal($a_{i/\theta_i}$, $e_{k_i/\theta_i}$ ))
**for all** rule number h in $P_o$ with the head $a_h$ such that *unify*($a_{i/\theta_i}$,$a_h$)$_{\theta_h}$ **do**
    delete($e_{h_0/\theta_i\theta_h}$).
**end for**
**for all** $a_{i/\theta_i}$ such that $P_t \models_{\theta_j} a_{i/\theta_j}$ **do**
    delete($e_{ki/\theta_i\theta_j}$).
**end for**

### 6.5.2 Comparison with Naive Approach

In the beginning of this section, we listed the following five tasks of the Naive approach that are optimized in the Optimized approach:

- Generating *EDBC* rules of a query when the query is registered.
- Generating substituted *EDBC$_s$* rules of the query.
- Adding *EDBC$_s$* rules to the knowledge base.
- Calling the *add*($f$) for all relevant facts in the knowledge base.
- Deleting *EDBC$_s$* rules (and *goal facts*) when the query is unregistered.

The Optimized approach optimizes these tasks using two techniques. The first technique is replacing $e_{k0}$ with a term constructed by the "$e_{k0}$" functor symbol and the arguments of $a$, as opposed to replacing it by *true*. This technique removes the need to perform the first three tasks and part of the fifth task. Using this technique, we only need to call the relevant *add*($e_{k_0}$) to register a query. The effect is that all relevant rules are automatically activated to generate the query results. The use of the $e_{k_0}$ term does not only provide a mechanism to activate relevant rules of a query $q$, thus removing the tasks 1, 3 and 5, but also applies a form of information passing by having the arguments of the query as its argument. In this way, the information about the bound arguments of the query is passed to the activated rules, limiting their computations to only generate the relevant results, the effect of which is similar to effect of the second task above.

Information passing is a technique used in bottom-up evaluation approaches such as *Magic Set* to limit the forward reasoning computations such that the only facts are derived that are relevant to a query [Beeri and Ramakrishnan, 1991].

The second technique is optimizing the fourth and fifth tasks. As explained in the previous section, the Optimized approach directly resolves sub-goals using the facts in the knowledge base and does not make separate copies of relevant facts for the sub-goals. Therefore, it removes the tasks of storing facts as *goal*($e_{k,i-1}$, $a_i$, $e_{k,i}$) goals and later on removing those *goals* when queries are unregistered.

## 6.6 Active Queries: Tabled Optimized Approach

In this section, we extend the Optimized approach with the *tabling* technique to make it more efficient and more declarative. To this end, we first briefly describe the *tabling* technique used in the paradigm of *Tabled Logic Programming* and then present the Tabled Optimized approach. Extending the Optimized approach with *tabling* relates our approach to research on incremental evaluation of *tabled logic programs*. We discuss the connection and presents empirical performance results.

### 6.6.1 Tabled Logic Programming

The *SLD* resolution in theory is complete in a sense that for every answer of a query $q$ to a program $P$, there is a *SLD* proof. However, the search for a proof in practice may not terminate, even when $P$ is a *datalog* program. *Datalog* programs are similar to definite logic programs, but with the restriction that arguments of terms can only be constants or variables.

For example, consider the query *path(0,Y)* to Program 4 presented in Listing 6.11. There are *SLD* proofs for both answers *Y=1* and *Y=2*. However, the query executed by *Prolog* never terminates and cannot produce any of these answers. *Prolog* resolves the query by applying the first rule leading to the sub-goal *path(0,Z)*. This sub-goal is resolved again by applying the first rule. The search falls into an infinite loop where infinite number of sub-goals being variants of the query itself are generated. Two sub-goals are variants if they can be made equivalent by renaming their variables.

```
1 path(X,Y) :- path(X,Z), edge(Z,Y).
2 path(X,Y) :- edge(X,Y).
3 edge(0,1).
4 edge(1,2).
```

LISTING 6.11: Program 4

The problem of vulnerability to infinite looping, for instance when there are left recursive rules as in Program 4, is well-known for logic programs. This problem can be partly addressed by the *tabling* technique studied in the paradigm of *Tabled Logic Programming* to make *Prolog* more declarative and more efficient [Swift and Warren, 2010]. The main idea in tabled logic programming is to cache the sub-goals encountered in query evaluation and the results of the sub-goals in tables. When a sub-goal is re-encountered, the cached results are used to resolve the sub-goal, rather than re-performing resolution against program clauses.

A well-known tabling mechanism is *SLG* resolution [Chen and Warren, 1996], an informal description of which is as follows. In *SLG* resolution, sub-goals encountered for the first time are resolved as in *SLD* resolution and their answers are cached. When a sub-goal is re-encountered, the cached answers are used to resolve the sub-goal and the sub-goal is suspended when all answers are used. The computation then continues by backtracking to explore some other computation path. Once more answers have been derived for suspended sub-goals, they are resumed and resolved against the new answers.

Tabling has a number of advantages [Swift and Warren, 2010] including the following. First, it ensures the termination of programs where the size of sub-goals and answers generated in query evaluation is less than a fixed number. Second, for a large class of programs such as *datalog* programs, tabling can achieve the optimal complexity for query evaluation. In general, tabling can factor out redundant evaluations of sub-goals by caching their results. However, tabling takes up computational and memory resources to create and manage tables and is beneficial only when sub-goals are repeated. In tabled logic programming systems, it is possible to declare only some of the predicates as tabled and non-tabled predicates are evaluated by *SLD* resolution.

#### 6.6.1.1 Incremental Evaluation of Tabled Logic Programs

When a query is evaluated, a tabled logic programming system tables the sub-goals. Not only can the cached results of sub-goals be used when sub-goals are re-encountered in the query evaluation, but they can be also used to resolve the sub-goals of other subsequent queries. However, the knowledge base may have been updated in the meantime by addition or deletion of facts (and rules) making the cached results stale. Approaches for incremental evaluation of tabled logic programs deal with the issue of how to update the answers of tabled sub-goals in accordance to changes of the knowledge base.

The major work on incremental evaluation of tabled logic programs is presented in the PhD thesis of D. Saha [Saha, 2006]. The work builds on top of the bottom-up evaluation approaches for *materialized view maintenance* in *deductive databases* [Gupta et al., 1993]. Two main approaches have been proposed. One is to determine the sub-goals whose results have been made stale due to knowledge base updates and re-evaluates them from scratch [Saha and Ramakrishnan, 2006a]. This approach is similar to the caching approach for agent programming languages [Alechina, 2013], mentioned in Section 6.1. The advantage of this approach is that it can be used for arbitrary *Prolog* programs that include, for instance, *Prolog* built-ins aggregation operations. The other one is to incrementally evaluate the changes of the tabled sub-goal results according to changes of the knowledge base which is more relevant to our work [Saha and Ramakrishnan, 2003].

### Incremental Evaluation of Tabled Logic Programs after Addition

For a definite logic program $P$ and a query $q$, let $ans_P(q)$ represent answers of $q$ with respect to $P$ and $\delta_P$ represent a set of facts and rules added to $P$. The problem is to find the smallest set $\triangle$ of answers such that $ans_{P \cup \delta_P}(q) = \triangle \cup ans_P(q)$. In other words, the problem is to find new answers of a query $q$ that can be inferred by considering the new set of facts and rules added to the knowledge base.

Given a program $P$ and an added program $\delta_P$, a transformed program $P'$ is generated as follows [Saha and Ramakrishnan, 2006a]. For each term $p/n$ (i.e. with predicate symbol $p$ and arity $n$) in $P \cup \delta_P$, a corresponding term $p'/n$ is introduced (i.e. $p$ is replaced by $p'$. The transformed program $P'$ include all clauses in $P$. For each fact $p$ in $\delta_P$, $p'$ is added to $P'$. For each clause of the form $a$:-$a_1 \wedge a_2.. \wedge a_n$ in $P \cup \delta_P$, the corresponding clauses $a'$:-$(a_1; a'_1) \wedge ..(a_{i-1}; a'_{i-1}) \wedge a'_i \wedge a_{i+1}.. \wedge a_n$ for each $i \in [1, n]$ is added to $P'$. The $i^{th}$ clause computes new answers of $a$ due to new answers of $a_i$. The transformed program $P'$ is such that $ans_P(q) \cup ans'_P(q') = ans_{P \cup \delta_P}(q)$. For example, Listing 6.12 presents the transformed program for incremental evaluation of addition to Program 4 presented in Listing 6.11.

```
1 path'(X,Y) :- path'(X,Z), edge(Z,Y).
2 path'(X,Y) :- (path(X,Z);path'(X,Z)), edge'(Z,Y)
3 path'(X,Y) :- edge'(X,Y).
```

Listing 6.12: Transformed Program for Incremental Addition to Program 4

The transformation is based on *finite differencing* [Paige and Koenig, 1982], widely used for materialized view maintenance in bottom-up query evaluation approaches [Gupta

et al., 1993]. The approach is also the underlying mechanism in the *Seminaive* bottom-up query evaluation [Ceri et al., 1990]. In the *Naive* bottom-up query evaluation, [9] rules are applied in a loop on the set of facts and derived facts until no new fact is generated. This approach is inefficient because each step applies the rules on all facts and derived facts so far. The *Seminaive* approach optimizes this by limiting the computation such that, in each step, only those facts are derived that depend on the set of new facts derived in the previous step.

### Incremental Evaluation of Tabled Logic Programs after Deletion

A well-known algorithm to update programs containing recursive rules after deletion of facts is *DRed* [Gupta et al., 1993]. *DRed* operates in three steps performed by bottom-up evaluation mechanisms. Given a program $P$ and a set of facts $\triangle$ to be deleted from $P$, the first step derives all facts that can be inferred from $P \cup \triangle$ that are depending on $\triangle$ and deletes them. The first step overestimates the set of facts to be deleted, because a fact which has a derivation based on deleted facts may have another derivation that does not include any of the deleted facts. The second step re-derives some of the facts deleted in the first step that have alternative derivations. Finally, the third step derives new facts using the facts re-derived in the second step.

D. Saha proposes an algorithm, that we call *Materialized DRed*, extending *DRed* as follows [Saha and Ramakrishnan, 2003]. For each derived answer, the facts and rules used to derive the answer are called a support of the answer and are maintained in a data structure called support graph. In *DRed*, the Seminaive approach is used to derive facts that are dependent on deleted facts. The support that causes the derivation of a fact for the first time is its primary support and is acyclic. When a fact is deleted, all its dependant facts are marked as deleted and the deletion propagates to other facts depending on the marked facts. However, an answer is not marked as long as it has an unmarked primary support. Taking the primary supports into account reduces the number of facts that are deleted and re-derived. The support graph takes up a lot of memory and has been made more compact in another work [Saha and Ramakrishnan, 2005]. In addition, another work has been presented that interleaves the deletion and addition such that an update operation is more efficient than the corresponding deletion and addition operations made separately [Saha and Ramakrishnan, 2006b].

[9] This is not to be mistaken with the *Naive* approach described in this chapter for active queries.

### 6.6.2 Tabled Optimized Approach

When evaluating the sub-goal $a_i$ of a binary rule of the form $e_{ki}$ :- $e_{k(i-1)} \wedge a_i$ in the Optimized approach, each rule whose head is unified with $a_i$ by applying a substitution $\theta$ is activated by the substitution $\theta$. We call such rules relevant-rules of the sub-goal. When evaluating the same sub-goal, the Optimized approach re-activates those rules again. Not only this has a disadvantage on performance, but also it could result in infinite looping. For example, evaluating the query $p(X)$ on a program that includes the rule $p$(X) :- p(X) and the fact *P(1)* using the Optimized approach falls into an infinite loop.

We adopt the tabling technique as follows. When a sub-goal is encountered for the first time, it is assigned a unique id and its relevant rules are activated by corresponding substitutions and all answers derived for the sub-goal by its relevant-rules are recorded. After this, when we evaluate a sub-goal of the same type (i.e. with the same predicate symbol and arity), we first check to see whether the cached results of the sub-goal derived by activating its relevant rules are re-usable. Relevant rules of a sub-goal are activated only when there is no prior sub-goal whose cached results can be re-used.

A sub-goal $i$ can re-use the cached results of a sub-goal $j$, if the sub-goal $j$ is a variant of the sub-goal $i$ or it subsumes it. Two sub-goals are variants, if there exists a renaming of variables that makes the two sub-goals equivalent and the sub-goal $j$ subsumes $i$ if there is a substitution that can be applied to $j$ to make it equivalent to $i$. Comparing among the two choices of variant or subsumption based sub-goal caching, both choices have their own advantage and disadvantage on performance. We have implemented both approaches, the choice of which is made by setting a parameter. When evaluating sub-goal $i$, if a sub-goal $j$ is found whose cached results could be re-used, we read those results and feed it to our algorithm as if these results were generated by activating relevant rules of the sub-goal $i$. Moreover, we record that the sub-goal $i$ depends on the sub-goal $j$ and whenever a new answer is generated by relevant-rules of the sub-goal $j$, we feed that answer to the sub-goal $i$ as well. However, if no sub-goal is found for re-use, relevant-rules of the sub-goal $i$ are activated with corresponding substitutions.

While the tabling mechanism above eliminates the unnecessary re-activation of sub-goals it does not guarantee the termination even when the size of sub-goals and answers generated in query evaluation is less than a fixed number. The algorithm falls into an infinite loop when there is a loop in the program that derives an answer based on itself. The simplest example is the loop made by the rule *p(X) :- p(X).* In general, there may be different ways to derive a single answer of a sub-goal. Resolving a sub-goal using the same answers multiple times is inefficient, in addition to bring amenability to infinite

looping. To deal with this issue, we implement the *answer invariant tabling* technique as follows. When an answer is found for a sub-goal, it is used to resolve the sub-goal, and other dependent sub-goals, only if it is a new answer and disregarded otherwise. An answer is new for a sub-goal, only if its invariant has not already been used and cached for the sub-goal. Another approach is to consider an answer new, only if it is not subsumed by another cached answer.

#### Incremental Evaluation

As described in Section 6.5.1, the Optimized approach naturally handles incremental addition. When a fact is added to the knowledge base, it is used to resolve all corresponding sub-goals as if the answer is derived by relevant rules of the sub-goals. The only difference in Tabled Optimized approach is that the added fact is first checked against the cached results of the matched sub-goals and is used to resolve a sub-goal only when its invariant has not already been used to resolve the sub-goal.

Incremental deletion is however more complex. When a fact is deleted, the Optimized approach derives its dependent derived facts and deletes them. In the Tabled Optimized approach however, an overestimated set of dependant derived facts may be deleted, due to the use of answer invariant tabling. The reason is that a fact which has a derivation based on the deleted facts may have a derivation that does not depend on the deleted fact and therefore should not be deleted. Consequently, a variant of the *DRed* algorithm is required. We implemented the *Materialized DRed* that explicitly maintains dependencies between derived facts and the rules and facts used to derive them [Saha and Ramakrishnan, 2003].

### 6.6.3 Evaluation

The purpose of this section is providing an empirical evaluation of the Tabled Optimized approach presented in this chapter for implementing active queries. The most relevant approach to the Tabled Optimized approach is tabled logic programming. In particular, the approaches for incremental evaluation of tabled logic programs are of the most interest to our work. While there are a few of such approaches, building on a large body of work on incremental view maintenance in deductive databases, the lack of standard benchmarks, open-source implementations and the application dependent performance of different approaches make it hard to provide a fair empirical comparison.

| task | path(0,X) | path(0,1000) | Add(edge(1000,1001)) |
|---|---|---|---|
| Xsb | 0.064 | 0.004 | 0.066 |
| Tabled Optimized | 2.047 | 0.051 | 0.014 |

TABLE 6.3: Comparision of performance on Program 5

To give some insight into the empirical performance of our implementation of the Tabled Optimized approach, we compare its performance with that of the *XSB* Prolog system[10] version 3.5.x for a number of query evaluation tasks on two programs. The reason for comparing our performance with that of *XSB* Prolog is that *XSB* offers one of the most stable and efficient tabled logic programming systems [Swift and Warren, 2010]. To evaluate the performance of the Tabled Optimized approach, we run the resulting transformed programs by the *SWI-Prolog* system[11]. The performances of the *SWI-Prolog* system, running the Tabled Optimized approach and the *XSB* Prolog system are evaluated on an XPS Intel Core i7 CPU@ 2.1 GHZ x 4 laptop running ubuntu 12.04 LTS.

Our first benchmark program is the reachability graph presented in Listing 6.13 where the graph has 1000 edges. Table 6.3 presents the query cost in seconds for the $path(0, X)$ and the $path(0, 1000)$ queries, and for the incremental addition of the $edge(1000, 1001)$ when the query $path(0, X)$ is active. For the *XSB* system, the predicate $path/2$ is tabled. In the case of the incremental addition for *XSB*, we first evaluate the query $path(0, X)$ and then add the $edge(1000, 1001)$ fact using the *XSB* $incr\_assert/1$ predicate and update the tabled results using the *XSB* $incr\_table\_update/0$ command.

```
1 path(x,y) :- edge(X,Z), path(Z,Y).
2 path(X,Y) :- edge(X,Y).
3 edge(0,1). edge(1,2). ... edge(998,999). edge(999,1000).
```

LISTING 6.13: Program 5

Our second benchmark program, presented in Listing 6.14 implements the following scenario. There are 100 artists, each is going to perform 30 shows. Each show has a price and a location. There are also 500 users each living in a location and is interested in 30 artists. For each artist of her interest, a user specifies the maximum price he would be willing to pay for a show of the artist and the maximum distance he would be willing to travel to see the show. The query $notify(X, Y, Z, W)$ computes which shows are of interest to who based on users' preferences. Table 6.4 shows the cost of the $notify(X, Y, Z, W)$ query and the update cost of adding the *loc(1,[2,3])* fact for the

[10] http://xsb.sourceforge.net/
[11] www.swi-prolog.org

| task | notify(X,Y,Z,W) | Add(loc(1,[2,3]) |
|---|---|---|
| Xsb | 4.61 | 4.64 |
| Tabled Optimized | 13.598 | 0.25 |

TABLE 6.4: Comparision of performance on Program 6

Tabled Optimized approach, executed by *SWI-Prolog*, and the *XSB* system. For *XSB*, the predicate $notify/4$ is tabled.

```
1  notify(Uers, Artist, Price, [LX,LY]) :-
2   show(Artist, Price, [LX,LY]),
3   likes(User, Artist, MaxPrice, MaxDistance),
4   Price < MaxPrice,
5   loc(User, [Lx,Ly]),
6   sqrt((LX-Lx)*(LX-Lx)+(LY-Ly)*(LY-Ly)) < MaxDistance.
7
8  loc(1, [6, 5]). ... loc(500, [2, 6]).
9
10 show(1, 10, [2, 8]). ...
11 ...
12 show(100, 2, [4, 2]). ...
13
14 likes(1, 33, 9, 13.237514377220565). ...
15 ...
16 likes(500, 74, 3, 5.851154344003308). ...
```

LISTING 6.14: Program 6

The performance results presented in Tables 6.3 and 6.4 show that the initial evaluation of queries to both programs are much faster by *XSB*. This is expected due to two reasons. The first reason is that the Tabled Optimized approach memorizes all intermediate results within the evaluation of each rule by binarization. However, in evaluation of Program 5, there are no intermediate results and the query evaluations in both approaches are similar in terms of the sub-goals and the results that are cached. The second reason, and perhaps the more important one, is due to the different data structures *XSB* and the Tabled Optimized approach use to cache and to search the sub-goal results.

The tabled Optimized approach is general in a sense that a transformed program (i.e. target program) is executable by any Prolog system. However, if the target *Prolog* system is known in advance, its indexing mechanisms can be taken into account to generate more optimized target programs. We only did one such optimization for *SWI-Prolog*, used to run our experiments, as follows. In our approach, each sub-goal is given a unique Id. The computed results of a sub-goal are cached by asserting *cached(id, result_k)* clauses where

each *result_k* represents a unique answer for the sub-goal *id*. Whenever a result is found for a sub-goal, it is propagated (i.e. used to resolve the sub-goal) only if it is a new one. To check if a result is new, it is checked against the cached results of the sub-goal. We optimized this operation for *SWI-Prolog* by indexing on the combination of the *id* and *result_k*. This was implemented by generating a hash key *HashKey_k* for each tuple *(id, result_k)* and recording cached results as *cached(HashKey_k, i, result_k)* clauses. This simple optimization made the transformed program of Program 5 more than 10 times faster. Such operations have been made highly optimized in systems such as *XSB* as result of long term developments and by a tight and low-level integration of tabling into the *Prolog* engine. For example, the use of *trie*-based data structures can significantly improve the performance of assertion and look-up of the cached answers [Ramakrishnan et al., 1999].

Regarding the incremental addition, it is observed that the Tabled Optimized approach performance is far more superior than that of the *XSB*. In fact, it is apparent from the results that *XSB* re-evaluates the queries from scratch, even though we use the *XSB* commands for incremental addition. Consequently, it seems that the current version of *XSB* implements the approach that invalidates and re-evaluates the cached results affected by updates. Nevertheless, the results show that incremental addition in Tabled Optimized approach is far more efficient than re-evaluation of queries from scratch.

To provide further insights into the performance of incremental addition in Tabled Optimized approach from the algorithmic point of view, we compares it with the approach based on *finite differencing*, discussed in Section 6.6.1.1. The Tabled Optimized approach memorizes all intermediate results of query evaluation within each rule. On the contrary, the *finite differencing* approach computes those intermediate results from scratch. The first consequence is that the Tabled Optimized approach uses more memory. The second consequence is that when the computations to derive intermediate results are more expensive than memorizing the results generated during the incremental addition, the Tabled Optimized approach is more efficient and vice versa.

For example, consider the incremental evaluation of the first rule of Program 6 when adding the fact *loc(1,[2,3])*. Listing 6.15 presents a program to derive new answers of the query *notify(X,Y,Z,W)* to Program 6 due to addition of the *loc(1,[2,3])* fact using the *finite differencing* approach[12]. To derive the new answers, the query *notify'(X,Y,Z,W)* to the program presented in Listing 6.15 is to be evaluated. Note that the incremental evaluation in *finite differencing* approach includes the evaluation of the clause *show(Artist,Price,[LX,LY]), likes(User,Artist,MaxPrice,MaxDistance), Price <MaxPrice.*

[12]The first rule includes more sub-goals to also account for addition of facts of type *show* and *likes*. As the example only considers the addition of a fact of type *loc*, those sub-goals have no effect and have been removed for brevity.

In the Tabled Optimized approach, the results of the evaluation of this clause is encoded and cached using *goal($a_i$, $e_{k_i}$)* facts. In our example, there are some *goal* facts each encoding that "there is a show by an artist that the user 1 likes and is comfortable with the price." The goal further encodes that "we are waiting for a new fact of type *loc(1,[$L_x$,$L_y$])* where the distance from the location *[$L_x$,$L_y$]* to the location of the show is less than an specified number and when such a fact is added to the knowledge base, a new fact of type *notify(X,Y,W,Z)* with corresponding substitution is to be generated." As *goal* facts *goal($a_i$, $e_{k_i}$)* are indexed by $a_i$ (in this example by *loc(1,[$L_x$,$L_y$])*), the relevant goal facts for addition of *loc(1,[2,3])* are efficiently accessed to derive the relevant *notify(X,Y,W,Z)* facts.

```
1  notify'(Uers,Artist,Price,[LX,LY]) :-
2   show(Artist,Price,[LX,LY]),
3   likes(User,Artist,MaxPrice,MaxDistance),
4   Price < MaxPrice,
5   loc'(User,[Lx,Ly]),
6   sqrt((LX-Lx)*(LX-Lx)+(LY-Ly)*(LY-Ly)) < MaxDistance.
7
8  loc'(1,[2,3])
9
10 loc(1, [ 6, 5]). ... loc(500, [ 2, 6]).
11
12 show(1, 10, [ 2, 8]). ...
13 ...
14 show(100, 2, [ 4, 2]). ...
15
16 likes(1, 33, 9, 13.237514377220565). ...
17 ...
18 likes(500, 74, 3, 5.851154344003308). ...
```

Listing 6.15: Finite Differencing Incremental Addition for Program 6

For incremental deletion of facts, we implemented the *Materialized DRed* approach described in Section 6.6.1.1. The materialization of the dependencies between facts and rules and derived facts and storage of the support graph, in our implementation turned out to be expensive. For example, the implementation of the approach changed the cost of the evaluation of the queries *path(0,X), Add(edge(1000,1001)) and notify(X,Y,Z,W)* from 2.047, 0.014 and 13.598 to 3.278, 0.019 and 20.035, respectively. The cost of *Materialized DRed* include the cost of deriving the dependencies and the cost of memorizing the dependencies. In our implementation of the Tabled Optimized approach, $e_{kn}$ (i.e. the head of the last binary rule of a clause includes $a_1$, .., $a_n$ (i.e. literals in the body

of the clause) as its arguments. Therefore, the dependencies are already accumulated and the only task to build the support graph is to parse the arguments of $e_{kn}$ into its separate elements. Therefore the main cost seems to be for storage of the support graph.

The Tabled Optimized approach caches all intermediate results within the rules when queries are evaluated. Consequently, it allows to efficiently compute the facts that can be derived from a fact, as described for the case of incremental addition. Therefore, the first and third steps of the basic *DRed* algorithm, which the *Materialized DRed* aims to optimize, can be efficiently implemented for the Tabled Optimized approach. The second step, checking if deleted facts have alternative derivations, can be also efficiently implemented querying the program for such facts and regarding the queries as tabled logic program queries instead of Tabled Optimized logic program queries. In other words, as we do not need to support incremental addition when performing the second step of *DRed*, a basic top-down tabled query evaluation strategy, without caching intermediate results within the rules, suffice. Therefore it seems that an approach to support incremental deletion, based on the basic *DRed* suits more for the Tabled Optimized approach. The implementation however is left as future work.

## 6.7 Related Work

The only robotic system that implements a form of active query mechanism is *ORO*. *ORO* does not however present any performance result for its implementation. In particular, the *Pellet* reasoning engine used by *ORO* does not support incremental evaluation of queries in general, for instance, when the knowledge base contains rules. After each change of the knowledge base, *ORO* re-classifies the whole knowledge and then evaluates the queries against the knowledge base, rather than updating only the results of the active queries as in the Tabled Optimized approach.

Some agent programming languages have recently adopted the idea of query caching to deal with performance issues caused by the repeated evaluation of queries on the agent knowledge base. Active queries takes the caching approach one step further by supporting incremental evaluation of queries. In the caching approach, cached results are invalidated and queries are evaluated from scratch when the knowledge base is updated with a relevant fact.

Among the existing logic-based knowledge representation and reasoning systems, approaches for incremental evaluation of tabled logic programs are of the most relevant to the Tabled Optimized approach presented in this chapter for implementing active queries. There are however subtle differences. The existing approaches provide a means

to declare sub-goals as tabled to cache their answers and incrementally update their answers. They do not however support registering and unregistering active queries. In these systems, the programmer manually manages what and when sub-goals are tabled. In our approach, all sub-goals of a query are tabled when the query is registered, they are updated by changes of the knowledge base as far as the query is registered and are disregarded when the query is unregistered. In other words, the management of what sub-goals should be tabled and updated according to active queries at the time is handled automatically.

From the technical point of view, the Tabled Optimized approach for incremental evaluation of sub-goals is similar to existing approaches for incremental evaluation of tabled logic programs. The difference is that the Tabled Optimized approach caches the intermediate results within rules that are computed during query evaluation using the binarization technique. Consequently, by supporting a more fine grained caching, it trades memory and performance of the first run of queries for better performance in incremental query evaluation.

Last but not the least, the Tabled Optimized approach is based on a source code transformation where the transformed program is executable by any *Prolog* system. Consequently, it can be used to bring the advantages of tabling and incremental query evaluation to well-developed Prolog systems such as SWI-Prolog that do not support tabling and incremental query evaluation. Developing a *Prolog* system with tabling and incremental tabling capabilities otherwise from scratch require a large amount of development effort. The implementation of tabling by applying source code transformation has been proposed before, for instance, by Rocha, R. [Rocha et al., 2007]. Existing transformation-based approaches however do not support incremental updating of queries.

## 6.8 Summary

In this chapter, we present two approaches for implementing active logic program queries. The Naive approach builds directly on top of the *EDBC* rules of the *ELE* system. The Optimized approach is a new approach developed in this thesis, improving over the Naive approach. The Optimized approach is then further extended into the Tabled Optimized approach using the tabling technique from the paradigm of *Tabled Logic Programming*. The Tabled Optimized approach transforms definite logic programs into *Prolog* programs with the following properties. First, the evaluation of a query on the transformed program, for instance, by a Prolog system that does not support tabling is performed in a top-down manner supporting tabling. Second, a component can register

queries to receive updates on their results when the knowledge base changes. Third, as far as a query is active, its results are incrementally and efficiently updated in a bottom-up manner when the program is updated by addition or deletion of facts. The incremental query evaluation approach is goal-directed in a sense that changes of the knowledge base are propagated such that queries of interest are updated and only the changes are propagated that are relevant for updating queries of interest.

The first advantage of the approach is that benefits of tabling such as avoiding infinite loops and reducing the time complexity can be added, for instance, to the well-known SWI-Prolog system that does not support tabling, without any need to change the underlying Prolog engine that would have been non-trivial needing a large development effort. Second, a fine grained level of caching is provided that supports a highly efficient incremental update of query results. Third, the approach can be incorporated in agent and robotic systems to support efficient implementation of active queries to be used, for instance, in plan execution and monitoring tasks enabling agents and robots to perform complex reasoning in dynamic environments.

# Chapter 7

# RobAPL Agent Programming Language

The question of this chapter is regarding the application of agent programming languages in robotics. We investigate what the plan execution control requirements of these language in robotics are and how to support the requirements. In particular, we are interested in the relation between such requirements and the information engineering requirements of these languages. We ask the question of whether and to what extent the information engineering techniques developed in this thesis address the information engineering requirements of these languages and support the real-time and event-driven execution of plans to apply these languages in robotics.

The chapter is organized as follows. First, we discuss the plan execution control requirements of these languages to facilitate their applications in robotics. Afterwards, we present the *RobAPL* language [Ziafati, 2014], a proposal developed in this thesis to extend agent programming languages with robotic plan execution capabilities. We will then follow by discussing the information engineering requirements of *RobAPL* and elaborate on how these requirements are addressed by *Retalis* and the information engineering techniques developed in this thesis. Finally, we discuss the related work and give a summary.

## 7.1 Plan Execution Control Requirements

Current agent programming languages (APLs) provide simple mechanisms for execution control of a robot's plans. Such mechanisms are often a combination of sequence, parallel, atomic, random order, ordered choice, random choice, conditional choice and

iteration plan operators. In order to facilitate the programming of autonomous robots, more advanced mechanisms are needed to deal with temporal and functional constraints related to a robot's tasks and its physics, to synchronize the parallel access of different plans to robot's resources and to handle the conflicts.

There are a number of robotic plan execution languages used to represent and execute plans. In these languages, plans are generated manually by robotic software developers or automatically by planning systems [Verma and Jónsson, 2005]. Such languages provide different mechanisms for controlling, coordinating and monitoring the executions of plans. In the following, we derive a list of robotic plan execution control requirements by generalizing from the functionalities supported by TDL [Simmons and Apfelbaum, 1998], PLEXIL [Tara and Vandi, 2006], APEX Freed [1998], SMARTTCL [Steck and Schlegel, 2010], PRS [Georgeff and Lansky, 1987] and PRS-lite [Myers, 1996] plan execution languages. To illustrate the requirements, we first present a usecase scenario.

### 7.1.1 Usecase Scenario

Araz and Mori are old and have Alzheimer. Moreover, Mori is under medication. To help them living easier and increase their safety, their children have bought them an assistant robot called NAO. NAO helps them by performing the following tasks:

1. T1: To remind Mori to take drug A every morning at 10 am. To remind Mori, NAO calls, "Take drug A Mori". When NAO hears the response back, "OK, I will take A", it considers the task as successfully finished.

2. T2: To check if drug A is finished. Drug A's color is red and is placed in a white box. NAO should check the box every afternoon to see if there are enough A in the box. If drug A is finished, NAO asks more if he has already ordered. Otherwise, it orders itself by sending an email to drugstore.

3. T3: To open the door if a visitor rings the bell. NAO checks the visitor face from the door camera; if it recognizes the face, it opens the door by pressing the OPEN_DOOR_BUTTON. Otherwise it informs Araz and Mori by calling, "A stranger is behind the door". In this case, the task is finished when NAO hears the response back, "Ok, I check it".

4. T4: To frequently check if there is any trash (a black cube) on the table and throw it to the trash can.

5. T5: To remind Araz and Mori about the places of their personal objects. E.g. Mori goes in front of the NAO's camera or introduces himself by saying, "It's

Mori" and asks NAO, "Remember my key is on the desk". He can later ask NAO, "Where is my key?". NAO should answer, "On the desk".

6. T6: To bring drinks from the kitchen table on users' requests.

7. T7: If Araz or Mori calls, "Help!", NAO should call for emergency assistance by pressing a RED_BUTTON placed in the the room. Also in T1 and T3, if NAO communicates with Araz/Mori for 3 times and does not hear the response back, NAO calls for emergency assistance as it might be sign of a dangerous situation.

### 7.1.2 Complex Plan Execution Control

To perform complex tasks, different plan operators are needed for synchronizing the execution of actions/plans in complex arrangements, beyond the simple sequential and parallel settings provided by the existing agent programming languages. For example to check whether there is enough drug in the box, NAO needs to go in front of the box (location L), orient its head's camera toward the box (orientation O), and then take and analyse a picture. To achieve this goal in an efficient way, NAO should be able to perform both actions of *move_to(L)* and *orient_head(O)* in parallel, and then to take a picture only after both *move_to(L)* and *orient_head(O)* actions have been successfully performed. Moreover it may be necessary for the camera to wait for a few second after the robot has arrived to the location and stopped walking, to stabilize before taking the picture.

Developing autonomous robot applications requires agent programming languages to be enriched with the following mechanisms to control and monitor the execution of plans.

- Composing a complex plan from a set of other plans (i.e. sub-plans) in sequence and parallel orderings in different levels of a hierarchy.
- Controlling and monitoring the execution of a plan at different levels of its hierarchy.
- Supporting conditional contingencies, floating contingencies (i.e. event driven task execution) and loops in the task tree decomposition. Different conditions are to be checked before, during and after the execution of a plan to control its execution.
- Supporting conditional and floating contingencies in monitoring the execution of a plan to guarantee its safe execution. Some conditions should be checked before starting/resuming the plan, some conditions should be checked continuously during the plan execution and some should be checked after finishing the execution of the plan.

- Governing and monitoring the plan execution (i.e. when to start, stop, suspend, resume or abort a plan/sub-plan) by different conditions such as temporal constraints on the absolute time and execution status of other sub-plans, occurrence of events, constraints on the state of the robot's environment and by direct access from other sub-plans (e.g. using shared variables).

### 7.1.3 Plan Execution Coordination

An autonomous robot has different goals and receives different events. In a BDI architecture, the robot generates different plans to achieve such goals and react to such events. To provide a good level of autonomy and intelligence, a robot should be able to follow its different plans in parallel. For example, when NAO is moving toward the drug box to check whether it's empty or not, it should be in the same time responsive to requests from its users (e.g. Task 5).

Execution of different plans in parallel can be conflicting due to a robot's functional and resource constraints and should be coordinated based on the priorities of plans. For example consider a use case in which NAO has picked up a piece of trash and going to put it into the trash can. Suddenly, NAO hears a user asking for help. To be able to help the user, NAO should go toward the Red_Button and have empty hands to press it. This plan has two conflicts with the previous plan of the NAO (i.e. walking to the trash bin and having trash in hand). As helping the user is of the highest priority, NAO should leave the trash and start walking toward the Red_Button immediately. In another case, a guest may ring the door. In this case, NAO should first put the trash it has in its hand into the trash can. It should then open the door and continue with the cleaning task afterwards.

To facilitate the use of agent programming languages for implementing control systems of autonomous robots, these languages should be extended with different mechanisms and corresponding programming constructs to support the coordination of parallel execution of plans. Moreover, the execution of plans should be monitored and their failures should be handled in a proper way. Plan execution coordination requirements include:

- Representing and determining conflicts between different plans (e.g. explicit representation by denoting the resources they require or by providing shared variables and locking mechanisms).
- Dealing with conflicts based on plans priorities and deadlines including dynamic prioritization and pre-emption.

- Supporting different policies to deal with pre-empted plans such as stopping, suspending or aborting.
- Recovering from a plan failure and performing wind-down activities after suspension and before resuming a plan.

## 7.2 RobAPL: a Robotic Agent Programming Language

In order to extend the plan execution capabilities of agent programming languages, we opt to build upon the *PLEXIL*[Verma et al., 2005, Gilles Dowek, César Muñoz and Pasareanu, 2010] plan execution language developed at *NASA* due to the following reasons. *PLEXIL* offers a simple structure for plan representation, a hierarchy of nodes with few syntactic constructs, but it is one of the most expressive plan execution languages unifying many of the existing ones. Moreover, *PLEXIL* has formal semantics which allows for the formal study of various types of determinism of plan execution. In addition, the operational semantics of *PLEXIL* is presented in a modular way at various levels of plan execution easing the formal study and modification of the language. Finally, the language has been successfully used in various robotic applications.

We adapt the *PLEXIL* syntax and semantics to be integrated in BDI-based agent programming languages for representing and executing plans. This includes introducing basic actions for querying and manipulating the agent's beliefs and goals in the BDI architecture and presenting an operational semantics for *PLEXIL*-like plan execution in the BDI deliberation cycle. Moreover, *PLEXIL* is extended to support pausing, resuming and pre-empting plans, performing clean-up and wind-down activities when pausing, resuming, aborting and pre-empting plans, and coordinating the parallel execution of plans over shared resources.

### 7.2.1 RobAPL Architecture

This section presents the *RobAPL* language for extending agent programming languages with PLEXIL-Like plan execution control capabilities. RobAPL architecture is based on a simple model of the existing BDI-based agent programming languages. It is designed to include the core components and operations of these languages and abstract away from their implementation details or specific features. The aim is to design an abstract language that can be adapted to extend the plan representation and execution capabilities of the existing BDI-based APLs to support their applications in Robotics.

RobAPL has the following components:

- A belief base and a goal base representing beliefs and goals of the agent.
- An event base representing events received from the outside world or generated internally by execution of the agent program during the last execution cycle.
- A plan base containing plans that are being executed by the agent.
- A rule base containing a set of plan generating rules that are applied to find a suitable plan for achieving a goal or responding to an event.

A plan generating rule specifies a partially instantiated plan that can be applied in a certain belief state to reach a goal, to respond to an event or to replace an *abstract action*. Such a plan is built upon the following basic types of actions.

- Belief-update: updating the belief base. A belief update action has a pre and post conditions. If the pre-condition is entailed by the agent belief base, the belief update action can be executed. The execution alters the belief base such that post-condition of the belief update action is entailed by the belief base.
- Goal-update: updating the goal base by adopting a new goal or dropping an existing one.
- External: performing an external action by invoking a function call. An external action can return a result value.
- Test: performing queries to the agent belief and goal bases to check whether the agent has certain beliefs and goals. If the action succeeds, it binds the free variables of the queries as result of performing the queries.
- Abstract: performing an abstract action which replaces this action with a plan that is associated to the abstract action by a plan generating rule.

In the rest of this section, we treat goals and events uniformly and call them events in the rest of this chapter, managed in the event base component. The difference between events and various types of goals in agent programming languages is in semantics of their dynamics. For example a goal of type achievement can be interpreted as a belief state that the agent wishes to brings about. In this case, if the execution of a plan for that goal fails, the goal is still in the goal set of the agent and is not removed from the agent goals. Our uniform treatment of event and goals supports the implementation of different semantics for events and goals. To support various semantics for events and goals, we assume that at the beginning of each deliberation cycle, some goals and events from previous deliberation cycles are added to the event base for example because the agent has not yet found suitable plans for them.

The RobAPL deliberation cycle consists of a planning and an execution step. In a planning step, plan generating rules are applied to plan for events in the event base. The result of this step is generation of a number of plans added to the plan base. In an execution step, events in the event base are processed again but this time for event-driven controlling of the plans. Events can be of a type for which a plan needs to be generated, of a type which is used for execution and monitoring of plans or it can be of both types.

### 7.2.2 RobAPL Plan Overview

A RobAPL plan consists of a hierarchical set of 8 types of nodes. Belief-update, goal-update, external, test and abstract are child nodes which are analogous to belief-update, goal-update, external, test and abstract actions, respectively. There are also list, resume/pause and abort/pre-empt parent nodes containing other nodes as their children. The root node of each plan is always a list node.

The execution of RobAPL plans (i.e. nodes) is controlled and monitored by a set of conditions on occurrence of events, the agent beliefs, the system time and a number of implicit and explicit attributes assigned to nodes. A node's attributes are the following ones among which the Id, priority, estimated execution time and resources are assigned by the programmer.

- Id: is a unique identifies of a node. Each node is uniquely identified by its own name and the name of its ancestors. The name of the list node at the root of a plan is randomly assigned at the run time.
- Status: represents the execution state of a node such as running, finished, etc.
- Outcome: represents the outcome of a node such as success, failed, etc.
- Execution Priority: is an Integer value used for resolving conflicts in parallel execution of nodes.
- Start time: indicates the system time at which a node starts execution.
- End time: indicates the system time at which a node finishes its execution.
- Estimated Execution time: is an estimated amount of overall time required for executing a node.
- Variables: containing all free and bounded variables used by a node.

- Resources: is a set of resource usages of the form $\langle Name, Type, Value \rangle$ where Name is a unique identifier of a resource, Type is one of the *blocking*, *using* and *adding* usage types and Value is an amount of resource usage.

The following are the conditions programmed for each node to control and monitor its execution.

- Start: determines when a node should start executing.
- End: determines when a node should stop executing.
- Invariant: determines when a node should abort executing.
- Pre: is checked right before executing a node and determines whether a node can start executing. If it does not hold, the node finishes its execution with the failure outcome.
- Post: is checked right after a node finishes its execution and determines whether the execution was successful. If it does not hold, the node finishes its execution with the failure outcome.
- Pre-empt: determines when a node should be pre-empted.
- Pause: determines when a node should pause executing.
- Resume: determines when a paused node should resume executing.
- Repeat: determines whether a node should repeat executing.
- Resource: determines when required resources of a node is available.

The pre, post and repeat conditions are queries on the agent's beliefs, the node's attributes, the system time and the status, outcome, start time and end time of other nodes of the same plan. These three types of conditions are only checked once when a node is going to start execution or it finishes its execution.

The start, end, invariant, pre-empt, pause and resume conditions are queries on occurrence of events and on the agent's beliefs, the node's attributes, the system time and the status, outcome, start time and end time of other nodes of the same plan. These conditions are continuously monitored during the time that they are allowed to make a transition in a node execution status. We will discuss the exact form of these conditions in Section 7.3.1. An agent has a pool of resources that nodes can query for resource availability. Moreover, the resource pool notifies the availability of resources when a node is waiting to acquire some resources.

### 7.2.3 RobAPL Plan Execution Operational Semantics

In the execution step of a deliberation cycle, plans are executed by processing all events of the event base in first-come first-served order by so called macro steps. In the beginning of a macro step, an event is processed making some conditions of some nodes in the plan base true. All such nodes make parallel and synchronous atomic transitions referred to as micro step. These transitions alter nodes' attributes which can make other conditions true resulting in another micro step. Micro steps are applied until no more micro step is possible. Detailed semantics of *RobAPL* atomic transitions are presented in Appendix B. The following informally describes the semantics.

The atomic transitions are defined in terms of atomic changes in execution status of individual nodes. At the beginning, all nodes are initialized in the Inactive state except the root node of each plan which is initialized in the Waiting state. In the Inactive state, none of the conditions of a node is monitored. A node in a Waiting state transits to the Executing state whenever its start condition becomes true, its pre-condition holds and its required resources are available. If the pre-condition does not hold, the node transits to the Iteration-Ended state having the Failure outcome. If required resources are not available, the node transits to the Waiting-Resource state from which it transits to the Waiting state again when resources become available.

Upon transiting to the Executing state, the action of a child node is executed which succeeds or fails. We assume all actions are performed in a synchronous way. By the synchronous execution, we mean that the next micro step is performed when actions of all child nodes in the Executing states are finished. For a Long running action which could long delay a micro step, the node can start the action by commanding an external component and then wait for the result to be received as an external event.

When the end condition of a node becomes true, an action node transits to Iteration-Ended state and its success or failure is determined by checking its post condition. Then if the repeat condition of the node is evaluated to true, the node repeats its execution by transiting from the Iteration-Ended state to the Waiting state. Otherwise it transits to the Finished state.

List nodes act as containers of other nodes. After a list node transits to the Executing state, its child nodes transit to the waiting state which are then monitored for execution. When the end condition of a list node becomes true, the list node does not immediately transit to the Iteration-Ended state but to the Finishing state waiting for its children being executed to finish their executions.

A node fails whenever one of its pre, post, invariant or pre-empt conditions is violated. A node also fails if one of its ancestors fails or a pause condition of one of its ancestors evaluates to true. When a failure occurs, action nodes in the Executing state abort their executions, list nodes being executed transit to the Failing state waiting for their children to be aborted and action and list nodes in inactive or waiting states skip execution. The outcome of a node specifies whether the execution of a node for the current iteration was skipped, successful or failure, whether it was failure of the node itself or its ancestors or whether it was due to the pre-emption of the node or its ancestors.

A node pauses its execution when its pause condition becomes true. The node first fails its children and then goes to the Paused state. When the resume condition of a paused node evaluates to true, it goes to the Resume state waiting for its resume nodes to finish executing and then transits to the Waiting state. When a node is paused, its children are put in the Inactive state if they were in the Inactive or Waiting state or if their repeat condition evaluates to true. Other children transit and remain in the Finish state.

The execution semantics of resume/pause and abort/pre-empt nodes are different than of the other types of nodes. These special types of nodes are for handling clean-up and wind-down activities when other nodes are paused, resumed, failed or pre-empted. The abort/pre-empt and resume/pause nodes transit from the Inactive state to the Waiting state when their ancestors are aborting/pre-empting or resuming/pausing. A list node which is failing/pre-empting or pausing/resuming waits for its abort/pre-empt or resume/pause children nodes to finish their executions before aborting or pausing/resuming its execution.

A difference between RobAPL and *PLEXIL* is the introduction of resume/pause and abort/pre-empt list nodes in RobAPL. The abort, pre-empt and pause list nodes are considered for execution before their parents are failed, pre-empted or paused. Similarly, resume list nodes are considered for execution before their parents are resumed. This facilitates a structured and bottom-up implementation of clean-up and wind-down activities for nodes that are failed, pre-empted or paused and support performing pre-resumption tasks before resuming nodes. Another difference is the distinction made between failing and pre-empting nodes in RobAPL to distinguish between execution failure, and pre-emption as the result of resource scheduling. This supports utilizing an external scheduler to monitor the plan execution to control pausing or pre-emption of plans based on their deadlines, priorities and available resources.

In each micro step, node transitions are performed in parallel and synchronously. There can be two sources of conflicts in parallel transitions of nodes. One type of conflict is when two nodes require a common resource of which is not enough available to be assigned to both. Similarly, access to shared variables and belief base and goal base

needs to be synchronized. For example, two test nodes could attempt to bind a shared variable to two different values. Whenever the execution of two nodes is conflicting, they are executed in the order of their priorities. The other source of conflict is when more than one transition is available for a node. Such conflicts are resolved based on priorities of transitions.

## 7.3 Information Engineering Requirements

In BDI-based agent programming languages, information from outside sources is received through a mix of active and passive perception mechanisms. In active perception, an agent receives information as the result of executing actions. After executing an action of a plan, the agent receives its result. The result is then used in the execution of the rest of the plan, for instance, to update the agent's goals and beliefs.

In passive perception, the agent receives information as events without taking explicit actions for it. At the beginning of each deliberation cycle, the agent processes the events received from its environment during the last deliberation cycle. In current BDI-based agent programming languages such as 2APL and GOAL, events are processed by means of event handling rules which generate plans in response. For example, event-handling rules in 2APL are of the form $\langle atom \rangle \leftarrow \langle belquery \rangle | \langle plan \rangle$. 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 may include external actions to react to the event and actions to update the agent's goals and beliefs.

The agent's knowledge of the environment in APLs is managed in the belief base (e.g. Prolog) and is updated through active and passive perceptions. This knowledge is queried by plan generating rules to generate plans for the goals and events and to choose and instantiate external actions during the plan execution. The interaction between the agent's belief base and the rest of the agent program is based on the request-response pattern of interaction where the belief base executes queries from the agent program and answers with the results.

Current APLs do not support on-flow processing of data due to their use of the request-response pattern for processing, management and querying of sensory data. While the on-flow processing functionalities can be implemented in current APLs using event handling rules, the lack of a systematic support for a high-level and event-driven implementation of on-flow processing functionalities makes the implementation difficult and inefficient for the following reasons.

- Concurrency: While deliberation in APLs is a cyclic process consisting of sense, reason, and act operations, on-flow processing is an event-driven process. Therefore on-flow processing functionalities 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 should be detected as soon as the last information (i.e. event) necessary for their detection becomes available. To this end, the belief base in current APLs should be continuously queried for events of interest after each update of the belief base. Also events should be kept in memory as far as they can contribute in the construction of an event of interest and removed afterwards. 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 and event-driven implementation of on-flow 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 on-flow 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.

- Ease of programming: implementing on-flow 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.

The lack of an event-driven (i.e. data-driven) and incremental query evaluation mechanism also results in performance issues in evaluation of the belief queries of plan generating rules. As explained in Section 6, an incremental query evaluation mechanism is needed to incrementally update the results of belief queries, as opposed to evaluate the queries from scratch.

Current agent programming languages also lack support for a high-level and efficient implementation of the on-demand processing functionalities to deal with the continuity, discreteness and asynchronicity of sensory data. For example, 2APL and GOAL use standard *Prolog* systems as their underlying knowledge base for managing beliefs. Consequently, they lack the on-demand processing supports added by the *SLR* language to *Prolog*.

### 7.3.1 Information Engineering in RobAPL

Our design choice to support information engineering in *RobAPL* is to support the development of separate Information Engineering Components (i.e. IECs) and their interactions with *RobAPL*, rather than tightly integrating information engineering support in the language. One reason is that, as argued above, on-flow 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 IECs and the robot's control component, implemented in *RobAPL*, supports the separation of concerns software engineering principle. Such a separation enables the development of re-usable IECs 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 is more than one control component. In such cases, the information generated and managed by an IEC can be used by more than one control component.

Moreover, enabling support for the development of IECs 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 information engineering languages for developing IECs as such languages evolve.

The *Retalis* language is a suitable choice to support information engineering in *RobAPL* due to the following reasons. First, it is a logic programming based language. and hence is easy to interface with BDI-based agent programming languages such as *RobAPL*. Second, it provide a comprehensive support for on-flow and on-demand processing and active queries unifying and advancing the information engineering functionalities of the existing systems.

As described in Section 7.2.2, the start, end, invariant, pre-empt, pause and resume conditions in *RobAPL* plans are queries on occurrence of events, the agent's beliefs and the node's attributes. These conditions should be continuously monitored when

they are allowed to make a transition in a node execution status. Using *Retalis* that integrates support for active queries as the underlying information engineering system of *RobAPL*, we can allow the implementation of these conditions by two mechanisms. The first mechanism is the implementation using event-rules. In this case, a condition is a complex event detected based on occurrence of a pattern of other events and a query on the *SLR* knowledge base of the *IEC* that processes the event rule. The second mechanism is the implementation using active queries. In this case, a condition is an active query on the knowledge base of an *IEC*. In both cases, as soon as the condition holds (i.e. the complex event is detected or the active query has an answer), the *IEC* informs the *RobAPL* program. In both mechanism, we assume that the information about the status, start time and end times of nodes are made available in the *IEC* knowledge base.

## 7.4 Related Work

The plan execution control capabilities of existing agent programming languages are very limited with respect to the requirements presented in Section 7.1. They provide limited support for parallel, event-driven and hierarchical task execution, synchronization and monitoring and then do not support the coordination of parallel execution of plans and implementing win-down activities. The information engineering capabilities of agent programming languages are also very limited. While there have been some attempts to implement robotic applications using BDI-based APLs [Ross, 2003, Verbeek, 2002], our research pioneers the systematic development of these languages for robotic applications.

While there are robotic plan execution languages that provide strong supports for the complex plan execution control requirements discussed in Section 7.1.2, these languages are often weak with respect to the plan execution coordination requirements discussed in Section 7.1.3. There is no such language that meets both sets of requirements. Furthermore, these languages are used to represent plans that are developed manually or by planners. Consequently, these languages lack the advantages of BDI-based APLs such as reasoning on goals and beliefs, reactive planning and plan repair capabilities.

The most close work to ours is the language of CRAM [Beetz et al., 2010] for robot programming. In CRAM, a knowledge base is provided to maintain and reason on the state of the environment which is similar to the belief base in APLs. Also in CRAM, plans are first citizen objects that can be manipulated and reasoned upon. However, the CRAM itself does not support the reactive planning and plan repair capabilities of BDI-based APLs. Moreover, the plan representation and execution language of CRAM provides no support for coordinating the parallel execution of plans over shared resources

and limited support for performing wind-down activities. Furthermore, the language comes with no formal semantics.

RobAPL extends the PLEXIL execution control and monitoring functionalities to monitor the availability of resources and control the pausing, resumption and pre-emption of plans. It also introduces new types of execution nodes to support the implementation of safety and wind-down activities in pre-emption, pausing and resumption of plans. While such monitoring and control functionalities provides a basic means for resource assignment and systematic pausing and pre-emption of execution nodes, we also need to incorporate a scheduling component for automatic scheduling of plans at run-time. To support the runtime scheduling of plans in, we will look into utilizing constraint satisfaction solvers such as EUROPA due to their expressive problem representation and any-time behavior. This requires developing a mapping from the representation of plans in PLEXIL into a constraint satisfaction problem for scheduling plans based on their estimated execution times, priorities and deadlines and their absolute and relative temporal orderings.

A related work is the PhD thesis of Fernando Koch [Koch, 2009] which investigates the requirements of BDI-based agent programming languages to implement *intelligent mobile services*. The BDI architecture is extended to support an efficient adaptation of the deliberation process according to situations of the environment in order to increase the responsiveness of the agent. This work shares many ideas and is complimentary to ours. The architecture supports an event-driven scheduling of goals and intentions, goals to which the agent has committed, according to their priorities. An observer module is presented to monitor events in order to efficiently determine when to process goals to generate plans and when to re-schedule the intentions, instead of checking all relevant conditions in every deliberation cycle. The agent's plans are executed in parallel threads that can be paused, resumed and aborted at runtime. We did not discuss the scheduling of goals. On the other hand, we presented a plan representation language that supports a complex event-driven control and synchronization of actions within a plan. The focus of the work of F. Koch is not on the plan representation and execution. More importantly, the observer module to a large extent remains conceptual and no general support for its implementation is provided. The on-flow processing and incremental query evaluation approaches presented in this thesis can be used to provide such support.

A feature of PLEXIL is its formal semantics that allows to analyse various properties of the language and PLEXIL plans. RobAPL extension to PLEXIL preserves the synchronous execution model of PLEXIL, but formal analysis of RobAPL integrating a PLEXIL-like plan execution control in a BDI-based deliberation cycle is left for future work.

## 7.5 Summary

The chapter presents work toward addressing plan execution control and information engineering requirements of agent programming languages to facilitate their use in robot programming. The PLEXIL language is adapted to be integrated in the BDI-architecture implemented by BDI-based agent programming languages. This includes introducing execution nodes for querying and manipulating agent's beliefs and goals and presenting a theoretical framework for interleaving the execution of PLEXIL-like plans with the plan generating phase of agent programming languages in each deliberation cycle of an agent.

RobAPL extends the PLEXIL language to support pausing, resuming and pre-empting plans and facilitating the implementation of clean-up and wind-down activities when pausing, resuming, pre-empting and aborting plans. Various future works are foreseen to mature the presented work. The proposed language should be implemented and used in practice to justify its usability for robot programming. Moreover, it is hard to manually verify whether the presented semantics follow the intuitions behind various operations of the language. It is also hard to manually verify whether various determinism properties of PLEXIL hold for the RobAPL language. However, similarity of RobAPL syntax and semantics to PLEXIL makes it amenable for formal analysis of its properties similar to formal analysis of PLEXIL.

# Chapter 8

# Summary

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

*Retalis* offers a high-level and declarative language for an efficient implementation of a wide range of information engineering functionalities. The requirements of *Retalis* are derived by generalizing from an extensive survey of research tasks related to robotic sensory data processing, management and querying. This includes an analysis of the functionalities supported by various classes of systems such as robotic frameworks, knowledge bases and active memories. *Retalis* advances the state-of-the-art robotic information engineering by integrating and extending the information engineering support of existing systems and approaches. It is evaluated by its detailed comparison with existing systems and empirical analysis of its performance.

The information engineering functionalities are classified into three models of information processing. On-flow processing is concerned with processing flows of data on the fly to detect complex events. On-demand processing is concerned with storing data in memory and querying it on-demand. The third model is concerned with incremental re-evaluation of queries referred to as active queries. Active queries are evaluated on-demand and their results are incrementally updated as new information is made available.

Robotic information engineering in its broad form includes all types of process that are performed in robot software such as recognizing faces in images. This thesis is concerned with a narrower form of processing data at system-level where data at higher levels of abstraction is provided by perception components and can be represented in symbolic form. In particular, the interest is on methods that support logical representation and reasoning in processing of data. Logic based approaches for system-level processing and management of data are dominant in robotics in order to integrate and reason about different pieces of common-sense and domain knowledge to empower robots with AI capabilities.

Previous work develops ontologies to model and represent data for service robots in house-hold task execution and human-robot interaction applications. In such applications, logic-based systems are used in robot software as central components for knowledge representation and reasoning. They enable ontological and logical reasoning and are interfaced with other components to support, for instance, spatial and probabilistic reasoning. Other work focuses on the integration and support of the processing and communication of information among the robot's software component.

This work provides a novel architecture for processing, management and querying of information in robot software. An information engineering component developed in *Retalis* processes its input flow of data on the fly and informs other components with information of their interests. It actively records relevant information and prunes the memory from unnecessary data. In addition to support querying the knowledge base, it also supports an incremental evaluation of queries. Consequently, an information engineering component is not a passive knowledge base that is updated and queried by other robot software. *Retalis* provides a high-level syntax to program the information engineering components and implements their functionalities efficiently.

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

## 8.1 Contribution

The contribution of this thesis is fivefold. The first contribution is the development of *SLR* language. *SLR* advances the state-of-the-art robotic on-demand processing systems by supporting a blend and extension of functionalities provided by active memories and logic-based knowledge management systems. On the one hand, histories of events are maintained in memory instances with unique identifiers. Similar to active memory systems, events are generated according to changes of the memory and histories are maintained according to some policies. On the other hand, logical queries can be performed on histories of data that are conveniently and efficiently accessed using some high-level language operators. In this sense, memory instances act like small knowledge bases whose content can be integrated and reasoned on. Consequently, a separate garbage collection profile can be defined for each memory instance. In addition, memory instances may contain knowledge that is not globally consistent but its pieces can be separately reasoned about.

The second contribution is the extension of *ELE* with a dynamic subscription mechanism and is integration with *SLR* languages concerning four issues. The first issue is to allow an external component to (un-)subscribe itself or other components to information processed by an information engineering component at runtime. Components can narrow down the information they receive by specifying a set of conditions on the information. Information can be filtered out based on the type of an event and a set of conditions including logical reasoning on its content and occurrence time. The second issue is to process flows of sensory data on the fly by *ELE* to extract relevant knowledge for its compact storage in *SLR*. The third issue is to query *SLR* for the knowledge built upon sensory data while processing flows of data. The fourth issue is to process events of changes of *SLR* memory by *ELE* to notify external components with patterns of changes that are of their interest.

The third contribution of the declarative *Retalis* language is a semantics based on a model of sensory data taking into account their occurrence times. This may be contrasted to alternative semantics based on processing times. In this way, the model captures and handles various issues related to asynchronous processing of data in robot software. In *ELE*, semantics of temporal relations among events are based on their occurrence times and the processing engine of the language correctly handles the cases where events are received unordered. *SLR* provides two mechanisms to synchronize queries over asynchronous events. Using these mechanisms, an IEC ensures to have received all relevant information from the perception components, before answering a query.

The fourth contribution is development of the *Tabled Optimized* approach for incremental evaluation of definite logic programs. This approach introduces a new way of interaction with logic programs. As opposed to the classical query-response interaction mechanism, a component can register a query and receive updates on results of the query. Changes to results of a registered query is incrementally computed as the knowledge base changes, significantly improving the performance over existing approaches.

The fifth contribution is design of the *RobAPL* language extending the plan execution support of existing agent programming languages to facilitate their applications in robotics. The proposal extends the *PLEXIL* plan execution control language with mechanisms for the coordination of parallel execution of plans and adapts it for plan execution control in BDI architecture. It is discussed that *Retalis* is suitable to support information engineering requirements of *RobAPL* to develop a BDI-based robot programming language.

Moreover, *Retalis* is an open-source[1] and framework-independent software library. Therefore, it can be used to empower the existing robotic frameworks with its wide range of functionalities as opposed to, for instance, robotic active memories which are tightly integrated with specific robotic frameworks. *Retalis* has been integrated in *ROS* and used to implement a few proof-of-concept tasks for NAO robot, including data transformation, runtime subscription, high-level event detection, sensory data management, state-based representation and query synchronization.

## 8.2 Conclusion and Future Work

The time has now come to conclude this work and review the questions laid down in introduction of this thesis. The following recalls and answers the questions and presents some directions of future work.

The overall question of this thesis is how to provide a language support for robotic information engineering. In autonomous robots with AI capabilities, logic-based representation is necessary to integrate and reason about knowledge from various sources. In particular, flexible action execution and human-robot interaction requires a formal representation and reasoning about common-sense knowledge rather than implicitly encoding such knowledge in control instructions of the robot that would lead to poor reusability and scalability. Consequently, we investigate information engineering methods that support logic-based knowledge representation and reasoning.

[1] The current release of *Retalis* does not include the support for active queries

Support of information engineering requires identifying and supporting general functionalities and design patterns that are useful in processing, management and querying of data in a wide range of robotic tasks. To this end, we classify the information engineering functionalities into three models of information processing. These models have been mostly the focus of separate research tasks. We derive the requirements of these models by an extensive survey of related work.

The requirements are not binary and different systems may support them to some extend. *Retalis* is developed that support all three models to a large extend integrating and advancing the existing approaches and systems. This is shown by a detailed comparison of *Retalis* with related work. Efficient implementation of information engineering functionalities is of a great consideration. We report a number of experiments showing the efficiency and scalability of *Retalis*.

Further evaluation and development of *Retalis* requires analysis of its application for information engineering in various robotic systems from different view points such as usability, performance, generality and comprehensiveness. It is hoped that making the *Retalis* open-source would encourage its use by the community to receive feedbacks essential for its further development.

An important question of information engineering is what kind of knowledge is relevant for robots and how to represent it. This question has been extensively studied in recent research on robotic knowledge management systems and therefore is not studied in this thesis. The W3C Web Ontology Language (OWL), based on description logics, is the common language for modelling knowledge in robotics. This language and tools for developing, maintaining and reasoning about OWL ontologies are being pushed forward by a large effort from the semantic web community and therefore are expected to remain as main technologies for knowledge representation and reasoning in robotics.

Logic programming systems such as *Prolog* used by *Retalis*, have been made mature over time and for reasoning purposes have some practical advantages compared to standard description logic reasoners. The advantages include a more compact knowledge representation by having the closed world assumption, a better support of reasoning about changes and actions by supporting a form of non-monotonic reasoning and easier integration of external functionalities and reasoning capabilities. An example of the latter one is the integration of the GML library for Mathematical computations in *Retalis* using the C++ interface of *SWI-Prolog*. While description logic reasoners are in general very efficient, a number of techniques are presented and developed in this thesis showing that *Prolog* can be used to process, manage and query a large flow of data on the fly and in fact may be a better choice even in term of performance. Anyway, description logics and logic programming have a large overlap in their representation expressiveness

[Lemaignan, 2012] and there are software libraries to manipulate OWL knowledge in *Prolog*, used for instance by the *KnowRob* system.

A short term future work of this thesis is to integrate existing robotic ontologies in *Retalis* and to develop the *RobAPL* language using *Retalis* as its information engineering language. Having these components in place provides us with a BDI-based cognitive framework for implementing control and information engineering components of autonomous robots that may be able to perform more complex task in dynamic environments and be more responsive comparing to existing robots.

Information engineering of autonomous robot that are to ground their knowledge on observations of their continuous and geometrical environments and to represent and reason about common-sense knowledge is challenging in many aspects and much remains to be explored in future research. For instance, we did not consider the uncertainty in representation and reasoning about the robot's sensory data. Most existing systems do not deal with uncertainty at the system and decision making level leaving it to lower level processing of data. With more advancement of formalisms and algorithms for representing and reasoning about uncertainty however, such support is to be integrated in information engineering. In the following, we discuss some future work regarding the sub-questions of this thesis.

### On-Flow Processing

This thesis suggests *ELE* as a suitable language for on-flow processing of information in robot software. We argue that the general requirements of robotic on-flow processing tasks are the same as the requirements for which on-flow processing languages such as *ELE* have been developed. On-flow processing is an emerging research field and yet formal comparison of expressiveness among existing languages is not available. However, existing languages are converging into supporting a set of operators to describe patterns of events and the supported operators provide a qualitative point of comparison. *ELE* is chosen as one of the most expressive on-flow processing systems and due to its support of logical reasoning on patterns of events.

A future work is to use *ELE* for processing and fusion of data in anchoring, situation recognition and plan execution control and monitoring tasks of different applications. The goal would be to understand and document the tasks that are convenient to develop by *ELE*. For other tasks, the language may need to be extended or revised to provide a better support while keeping its efficiency. In particular, various forms of consumption

policies are often convenient and even necessary to implement on-flow processing functionalities. *ELE* supports a few that change the semantics of the language in an ad-hoc way. More comprehensive and systematic support of such policies is to be investigated.

Event rules can be added or removed from *ELE* at runtime. In addition, robot software components can subscribe to events from *ELE* and *ELE* can be subscribed to events from other components at runtime using the *Retalis* interface of *ELE*. An interesting future work is to make such configurations automatic according to, for instance, the plan execution context of the robot. For example, if an event is of interest in a given context, the event rule required for its detection could be automatically recognized, parametrized and added to *ELE* and *ELE* could be automatically subscribed to events that are relevant for the rule. Such automatic adaptation of processes have been developed for the DyKnow framework.

### On-Demand Processing

*SLR* extends *Prolog* with domain specific operators to manage and query the asynchronous and discrete flows of sensory data in the knowledge base. The *SLR* design has aimed for simplicity and efficiency supporting memory management and state-based representation at rather low levels of implementation. Further models and mechanisms are needed to provide higher level support for a compact and efficient representation, storage and querying of the robot's knowledge built upon its sensory data, some outlined below.

A future work is to integrate and extend existing robotic ontologies. Moritz Tenorth, the developer of the *KnowRob* ontologies points out the need for a more thorough representation of spatial information including semantic representation of units of measure, coordinate frames and transformations among them. In a more broad sense, Severin Lemaignan, the developer of *ORO* ontologies, points out the need for a more proper context management. This includes identifying what contextual information is relevant for robots and how to represent, store and reason about it. For instance, contextual information could be attached to the facts in the knowledge base or groups of facts could be modularized according to different context. Another future work pointed out by Severin Lemaignan is management of inconsistent knowledge. Simply recognizing inconsistent knowledge that, for instance, may arise due to error in perception is not enough. Mechanisms are required to solve and remove inconsistency that would otherwise prevent reasoning about the knowledge.

Another future work is to further support compact storage of information in memory. In *Retalis*, the *ELE* language can be used to extract information from the input flows of

data in order to make the information stored by *SLR* more contact. As pointed out by Motitz Tenorth, an effective way to reduce memory usage is to compute expectations and store only surprising data. *Retalis* supports this by querying *SLR* in on-flow processing. In general, the granularity at which to record changes of the robot's environment is to be determined and mechanisms to compute and efficiently store the changes is to be investigated. In addition, mechanisms for removing information about the past into external long-term memories and accessing it when necessary may be required.

*SLR* supports state-based representation for instance to deal with temporal validity of knowledge using *next* and *prev* operators. While these operators are simple and efficient to reason about changes of the robot's state of the environment, a more thorough support of reasoning about changes of the robot's state is required. In particular, mechanisms are required to specify and reason about the relation between fluents, describing state of the world, and actions and events, as studied in logical formalisms such as situation calculus and event calculus.

### Active Queries

Tabled Optimized approach supports the implementation of active queries which are definite logic program queries. A short term future work is on incremental update of query results after deletion of facts as well as developing more optimized data structures for caching and accessing sub-goal results. A long term future work is to extend the Tabled Optimized approach to support the negation as failure operator in logic programming. In addition, supporting built-in *Prolog* predicates such as aggregation or random predicates is other part of future work.

### RobAPL Agent Programming Language

This thesis presents the design of *RobAPL* language for a BDI-based implementation of control components of autonomous robots with advanced plan execution control capabilities. The implementation of the language, as well as its integration with *Retalis* to support information engineering in planning and plan execution is however left as future work.

# Appendix A

# Retalis API and Tutorial

This chapter presents the API of Retalis and provides a tutorial on implementation of the NAO application presented in section 6.

## A.1 Nodes

The *Retalis ROS* package includes two *ROS* nodes: *retalis* and *retalis_ros_interface*.

The *retalis* node is the main one, implementing on-flow and on-demand functionalities. This node provide the following services to configure the *Retalis* at runtime.

- *add_output_subscription*: adds a subscription.
- *delete_output_subscription*: deletes a subscription.
- *add_memory*: adds a memory instance.
- *delete_memory*: deletes a memory instance.

The *retalis_ros_interface* automates the conversion between *ROS* messages and *Retalis* events. It provides the following services to configure the *Retalis* at runtime.

- *add_input_subscription*: subscribes *Retalis* to a *ROS* topic.
- *delete_input_subscription*: un-subscribes *Retalis* from a *ROS* topic.

## A.2 Tutorial: Coordinate Transformation for NAO Robot

The usecase is to position the recognized objects in the world coordination frame and reason on high-level events occurring in the environment.

Input data to *Retalis* are:

- Recognized objects (ar_pose markers) positioned relative to the robot's top camera
- Recognized faces
- Transformation among coordinate frames

Output data from *Retalis* are:

- Position of objects in the environment
- Events about situations of the environment

### A.2.1 Programming

*Retalis* is programmed using three files, located in the *application_source* folder of the *Retalis* package:

- *pub_sub.xml* : subscribes *Retalis* to *ROS* topics. It also specifies the message types of topics to which *Retalis* may publish messages.
- *goalPredicates.txt*: creates a set of subscriptions and memory instances. A subscription subscribes a *ROS* topic to *Retalis* by a policy to selectively send data to that topic. A memory instance selectively records and maintains data in the *Retalis* knowledge base based on a policy.
- *eventRules.txt*: specifies rules to perform complex event-processing functionalities and to query the *Retalis* knowledge base.

### A.2.2 Input from *ROS*

*Retalis* is subscribed to the *tf*, *ar_pose_marker* and *face_recognition/feedback* topics as in Listing 2.2, see the *pub_sub.xml* file. Messages received by *Retalis* from the subscribed topics are automatically converted to events. For example, Figure A.1 presents a message of type *tf/tfMessage*. This message is converted to the *Retalis* event, presented in Figure A.2.


FIGURE A.1: Example of a *tf/tfMessage* message

```
tf__0__tfMessage([
        geometry_msgs__0__TransformStamped(
                        std_msgs__0__Header(0, [1413748205, 981209993], '"/odom"'),
                        '"/base_link"',
                        geometry_msgs__0__Transform(
                                        geometry_msgs__0__Vector3(0,-0.01,0.1),
                                        geometry_msgs__0__Quaternion(0.05,0.008,-0.02,0.9)
                                                )
                                        )
                ])
```

FIGURE A.2: Example of a *tf/tfMessage* event

### A.2.3 Event-Processing

Listing A.1 shows a part of the *eventRules.txt* program that splits each *tf/tfMessage* event into a set of events by calling the *split_tf* function. The function is implemented as a *Prolog* rule. It splits the *Transforms*, its input list of *geometry_msgs/TransformStamped* messages, into the elements and generates a new *tf* event for each. Each *tf* event is time-stamped according to header file of the corresponding message.

```
null do split_tf(Transforms) <- tf__0__tfMessage(Transforms).

split_tf([Head|Tale]) :-
  Head = geometry_msgs__0__TransformStamped(
           std_msgs__0__Header(_, [S, NS], Parent),
           Child,
           geometry_msgs__0__Transform(
                    geometry_msgs__0__Vector3(P1,P2,P3),
                    geometry_msgs__0__Quaternion(Q1,Q2,Q3,Q4)
                                                )
                                                    ),
  new_event(tf(Parent,Child,[P1,P2,P3],[Q1,Q2,Q3,Q4],S,NS),
  split_tf(Tale).

split_tf([]) .
```

LISTING A.1: *Retalis* Splitting *tf/tfMessage* events

For instance, when *Retalis* receives the event presented in Figure A.2, the following event is generated:

```
tf('"/odom"', '"/base_link"',[0,-0.01,0.1],
   [0.05,0.008,-0.02,0.9])
```

LISTING A.2: Example of *tf/tfMessage* event

The event is time-stamped with the time-stamp (1413748205, 981209993), represented in the *datime* format. The format of time-stamps are *datime(Y,M,D,H,Min,S,Counter)* where *Counter* encodes nanoseconds since seconds (i.e. *stamp.nsec* in *ROS* messages).

### A.2.4 Memorizing

The following clause in the *goalPredicates.txt* file creates a memory instance, named *odom_base*, that keeps the history of the last 2500 events of the form *tf( '"/odom"','"/base_link"',V,Q)*. The list of conditions on events to be recorded is empty. Events are recorded in the *tf(V,Q)* format.

For example, the event presented in Listing A.2 matches this memory instance. This event is recorded by this memory instance as *tf([0,-0.01,0.1], [0.05,0.008,-0.02,0.9])*.

### A.2.5 Querying

The Prolog program in the *eventRules.txt* file together with the dynamic knowledge maintained by memory instances represent a *Prolog*-based knowledge base. Queries to the knowledge base are normal *Prolog* queries, but with two main differences, described below.

#### A.2.5.1 Accessing Memory Instances

An example of using prev and next terms is presented in Listing A.3. The input values to the *interpolate_tf(Id,T,Pos)* function is the Id of a memory instance, keeping the history of some *tf* events, and a time point. The *tf* events represent observations of the transformation between two coordinate frames over time. From these observations, this function interpolates the transformation between the frames at time $T$. This is implemented as follows. The last observation before $T$ and the first observation after $T$ are found using the prev and next terms. Then the position is linearly interpolated by making a function call to the *OpenGL* Mathematics library that has been integrated with *Retalis*.

```
1 interpolate_tf(Id,T,Pos):-
2         prev(Id, tf(V1,Q1), T1, _, T),
3         next_inf(Id, tf(V2,Q2), T2, _, T),
4         datime_interpolate(T1,T2,T,Fraction),
5         interpolate_quaternion(V1,Q1,V2,Q2,[Fraction],Pos).
```

Listing A.3: interpolate_tf function

The *transform_marker(RelativePos,Time,AbsolutePose)* function in Listing A.4 uses the *interpolate_tf* function to position an object in the world reference frame. Given *RelativePos*, the position of an object relative to the camera at time *Time*, this function computes *AbsolutePose*, the position in the world, as follows. First, it changes the time format from *ROS* time to *datime*. Then, it interpolates the transformation between */odom*-to-*/base_link*, *base_link*-to-*torso*, *torso*-to-*Neck*, *Neck*-to-*Head* and *Head*-to-*CameraTop_frame* at the time *Time*. Third, it applies these transformations on the *RelativePos* by making a function call to the *OpenGL* Mathematics library. It is assumed that the */odom* frame is aligned with the world reference frame.

```
1 transform_marker(RelativePos,Time,AbsolutePose):-
2                 convert_to_datime(Time,T),
3                 interpolate_tf(odom_base,T,P1),
```

```
4               interpolate_tf(base_torso,T,P2),
5               interpolate_tf(torso_neck,T,P3),
6               interpolate_tf(neck_head,T,P4),
7               interpolate_tf(head_cam,T,P5),
8               transform_quaternion([P1,P2,P3,P4,P5,RelativePos],
9                                    AbsolutePose).
```

LISTING A.4: transform_marker function

#### A.2.5.2 Query Synchronization

To interpolate the position, for instance, between the */odom* and */base_link* coordination frames at time $t$, the position should have been observed, at least once, after $t$. The observations, *tf/tfMessage* messages here, are received asynchronously. Therefore, the *interpolate_tf(odom_base,t,Pos)* function should be evaluated only after the *odom_base* memory instance has been updated with an event occurring after $t$. This is realized in *Retalis* using a synchronized event, as follows. The *synchronized(Event,Query,SynchConditions)* function, performs the *Query*, when the *SynchConditions* are satisfied and then generate the *Event*. An example of a synchronized event rule is presented in Listing A.5. This rule computes the position of recognized markers in the world and is read as follows.

```
1  syncronized(
2   geometry_msgs__0__PoseStamped(
3         std_msgs__0__Header(Seq, [Sec,NSec], Name),
4         geometry_msgs__0__Pose(
5                   geometry_msgs__0__Point(P11,P12,P13),
6                   geometry_msgs__0__Quaternion(Q11,Q12,Q13,Q14)
7                                       )
8                                          ),
9   transform_marker(
10        RelPose,[Sec,NSec],[[P11,P12,P13],[Q11,Q12,Q13,Q14]]
11          ),
12  [[odom_base,Z],[base_torso,Z],[torso_neck,Z],
13     [neck_head,Z],[head_cam,Z]]
14              )
15  <- ar_marker(Seq,Name,RelPose,Sec,NSec)
16     where (Z is Sec + (NSec * 0.000000001) ).
```

LISTING A.5: Synchronized event rule

For each *ar_marker* event, specified in Line 15, the position is computed by calling the *transform_marker* function, as in Line 9. After computing the position, a *geometry_msgs__0__PoseStamped* event, as in Line 2, is generated. Such an event encodes the marker's name, its position in the world and the time of recognition.

The *SyncConditions* are specified in Line 12. These conditions specify that the *transform_marker* function should be evaluated, only after all *odom_base, base_torso, torso_neck, neck_head* and *head_cam* memory instances have been updated, at least once, with events occurring after time $Z$. The time $Z$ is the time of recognition of the marker.

*Retalis* performs the synchronization of events in an event-driven and efficient way. The generation of a synchronized Event is postponed, until the *SyncConditions* are satisfied. Postponing an event does not postpone the generation of other events and postponed events are generated as soon as necessary conditions are met.

### A.2.6 Subscription

Listing A.6 presents a subscription clause from the *goalPredicates.txt* file. The subscription subscribes the topic *marker1* to the *PoseStamped* events in which *Name* is *'"4x4_1"'*. Such events are generated by the synchronized event rule, presented in Listing A.5. They contain the position of the marker *4x4_1* in the world coordination frame. The id of the subscription is $m1$ which can be used to cancel the subscription at any time.

```
1 subscribe(
2   geometry_msgs__0__PoseStamped(
3         std_msgs__0__Header(_,_,'"4x4_1"'),_), [],
4   _,
5   marker1,
6   m1)
```

Listing A.6: Subscription

# Appendix B

# *RobAPL* Plan Execution Atomic Transitions

Figures 1-17 present semantics of *RobAPL* plan execution atomic transitions in similar notations to transition diagrams of Plexil [Tara and Vandi, 2006] as follows. The eclipses represent node states. The rectangles represent condition changes that cause a transition from a node state. Only the condition change explicitly represented causes the transition. The diamonds represent checks and the hexagons represents node outcomes. Transitions are represented by directed arrows. If multiple transitions are simultaneously enabled, the top-down order of presenting transitions represent the precedence order. The T, F and U represents the evaluation of a condition to true, false and unknown. The abbreviations Inv, Prmt, P-failure and P-Prmt correspondingly represent Invariant and Pre-empt conditions and Parent-Failure and Parent-Pre-empt outcomes. The resume/pause and abort/pre-empt nodes are called wind-down list nodes.

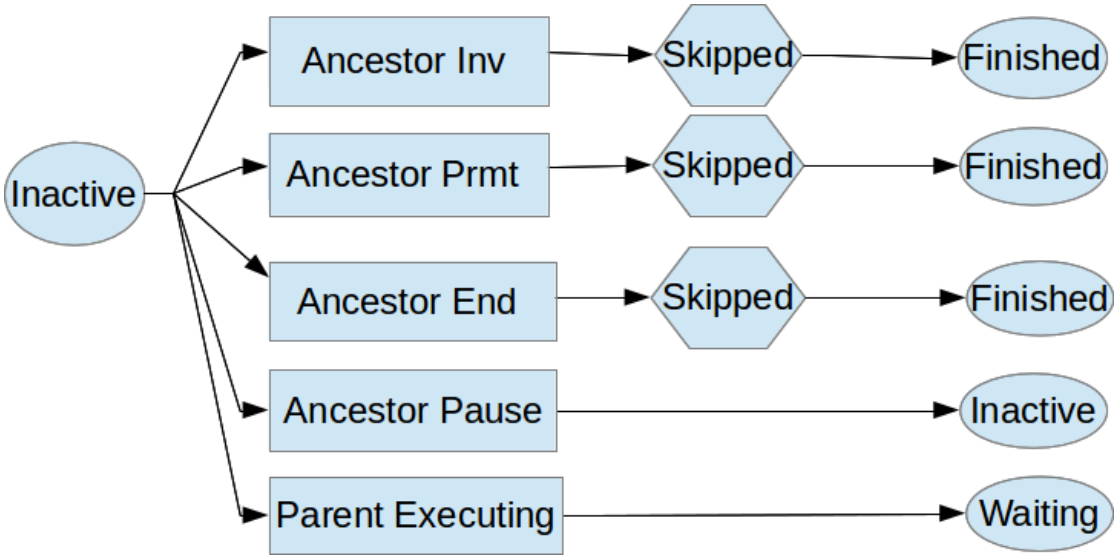


FIGURE B.1: Transitions of child and list nodes from the Inactive state

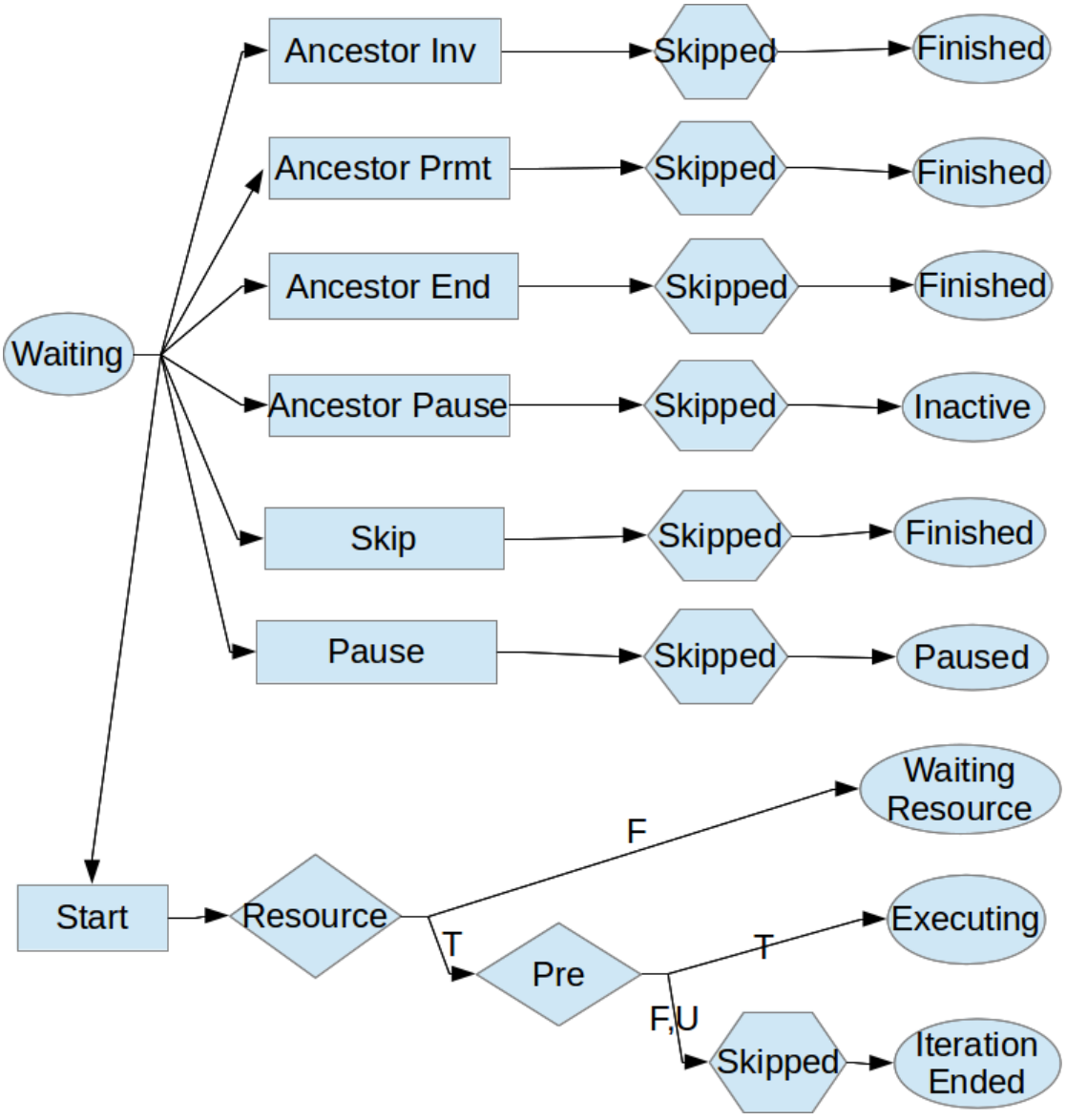


FIGURE B.2: Transitions of child and list nodes from the Waiting state

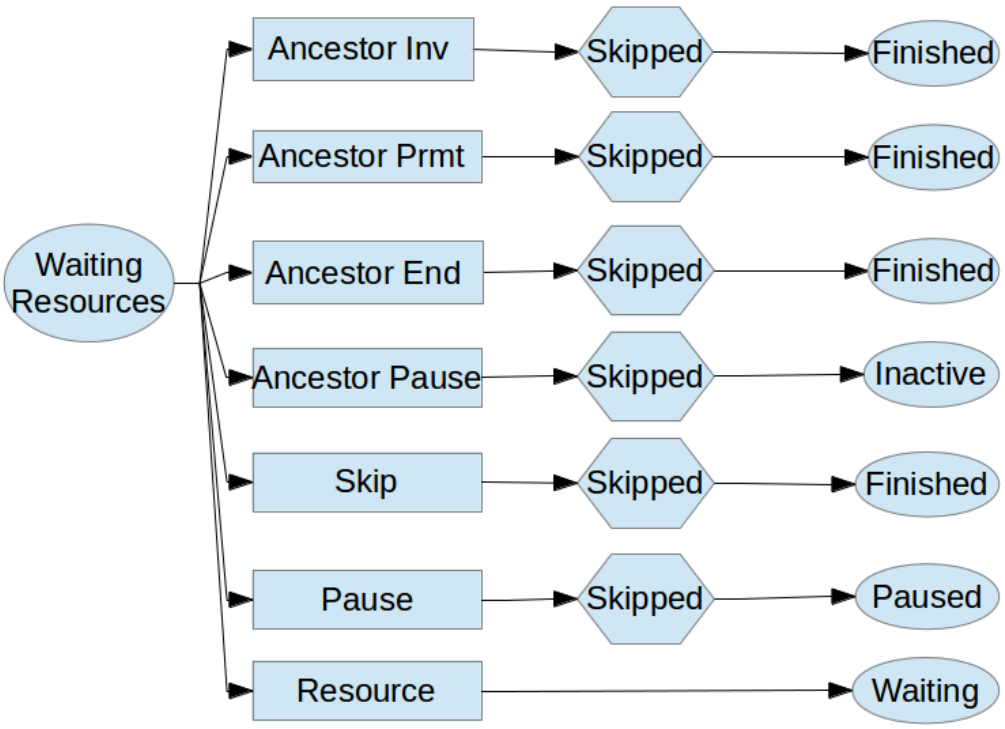


FIGURE B.3: Transitions of child and list nodes from the Waiting-Resources state

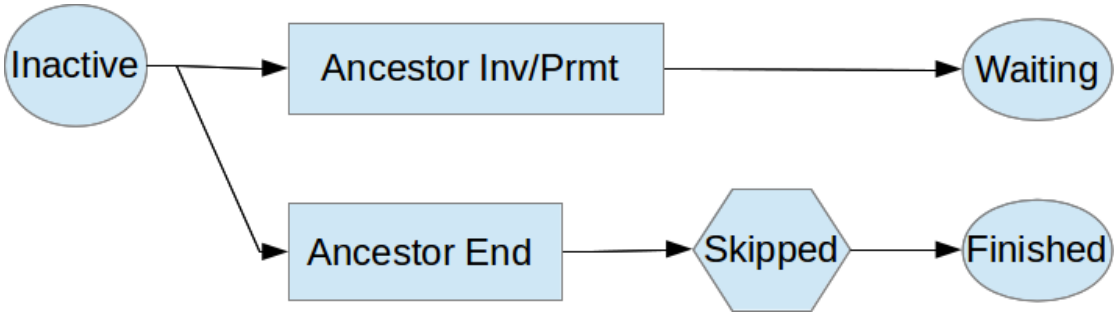


FIGURE B.4: Transitions of abort/pre-empt nodes from the Inactive state

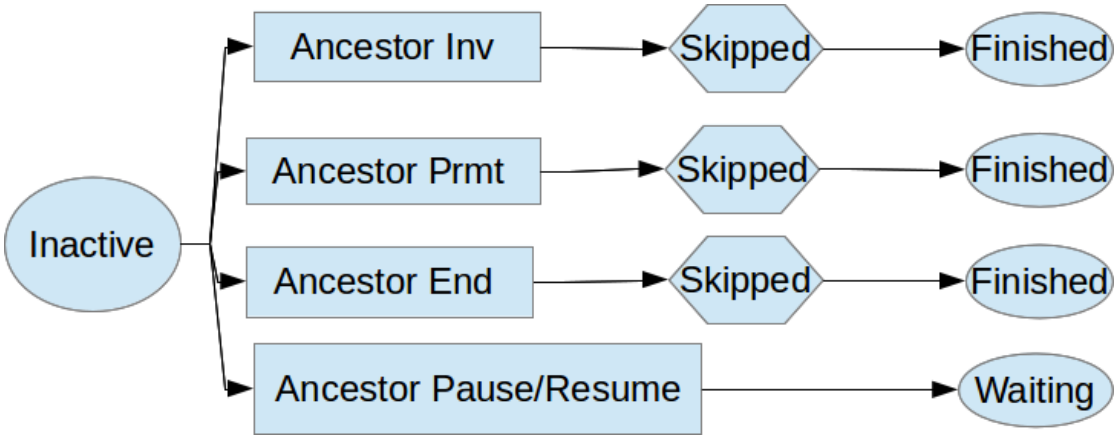


FIGURE B.5: Transitions of pause/resume nodes from the Inactive state

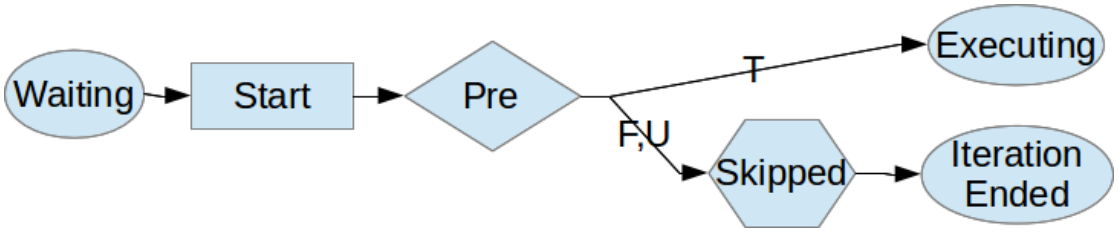


FIGURE B.6: Transitions of abort/pre-empt nodes from the Waiting state

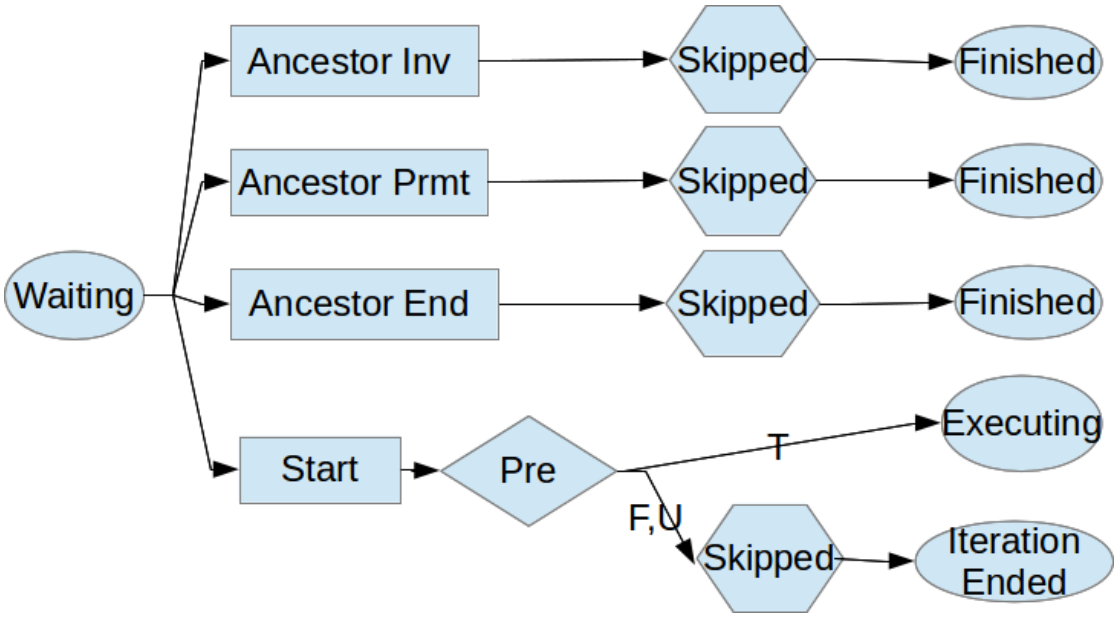


FIGURE B.7: Transitions of pause/resume nodes from the Waiting state

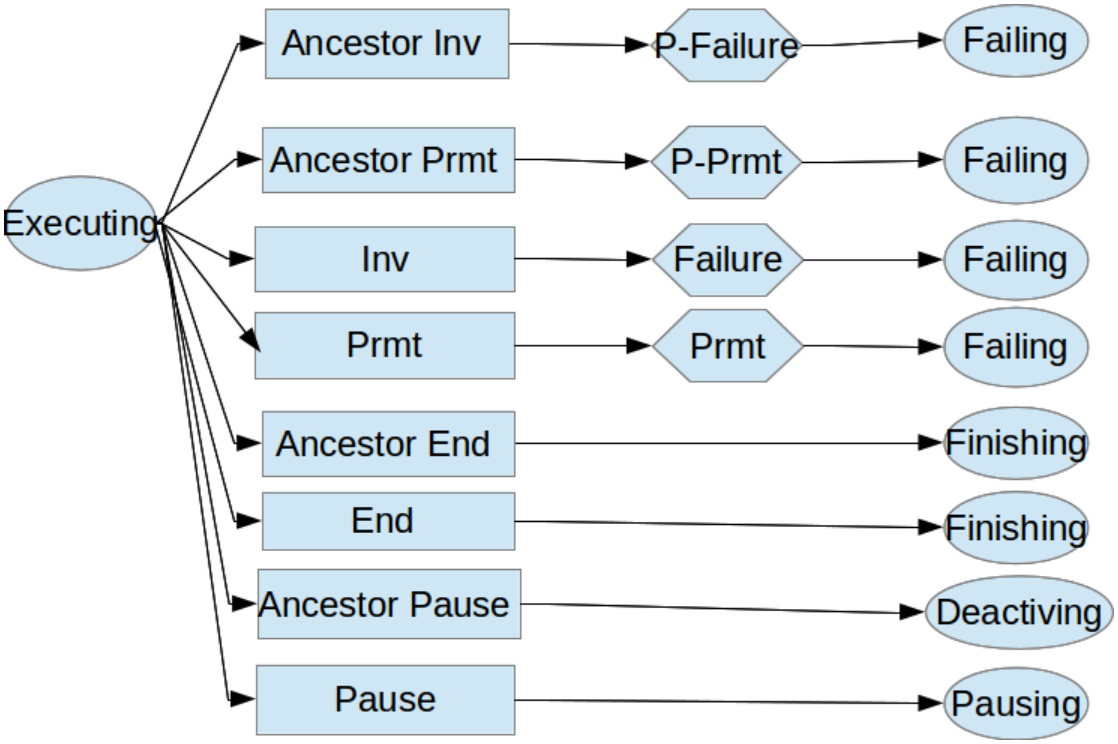


FIGURE B.8: Transitions of list nodes from the Executing state

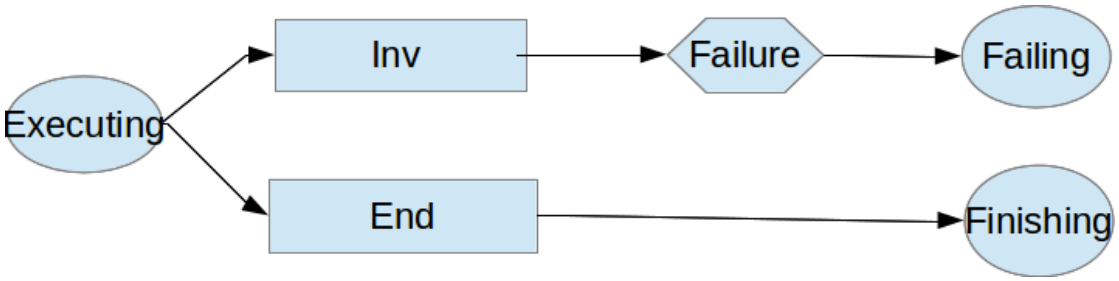


FIGURE B.9: Transitions of abort/pre-empt nodes from the Executing state

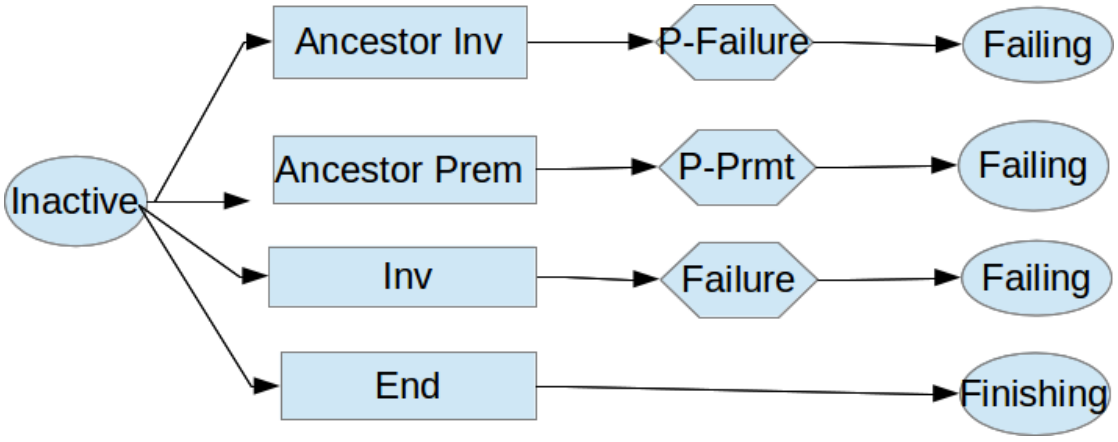


FIGURE B.10: Transitions of pause/resume nodes from the Executing state

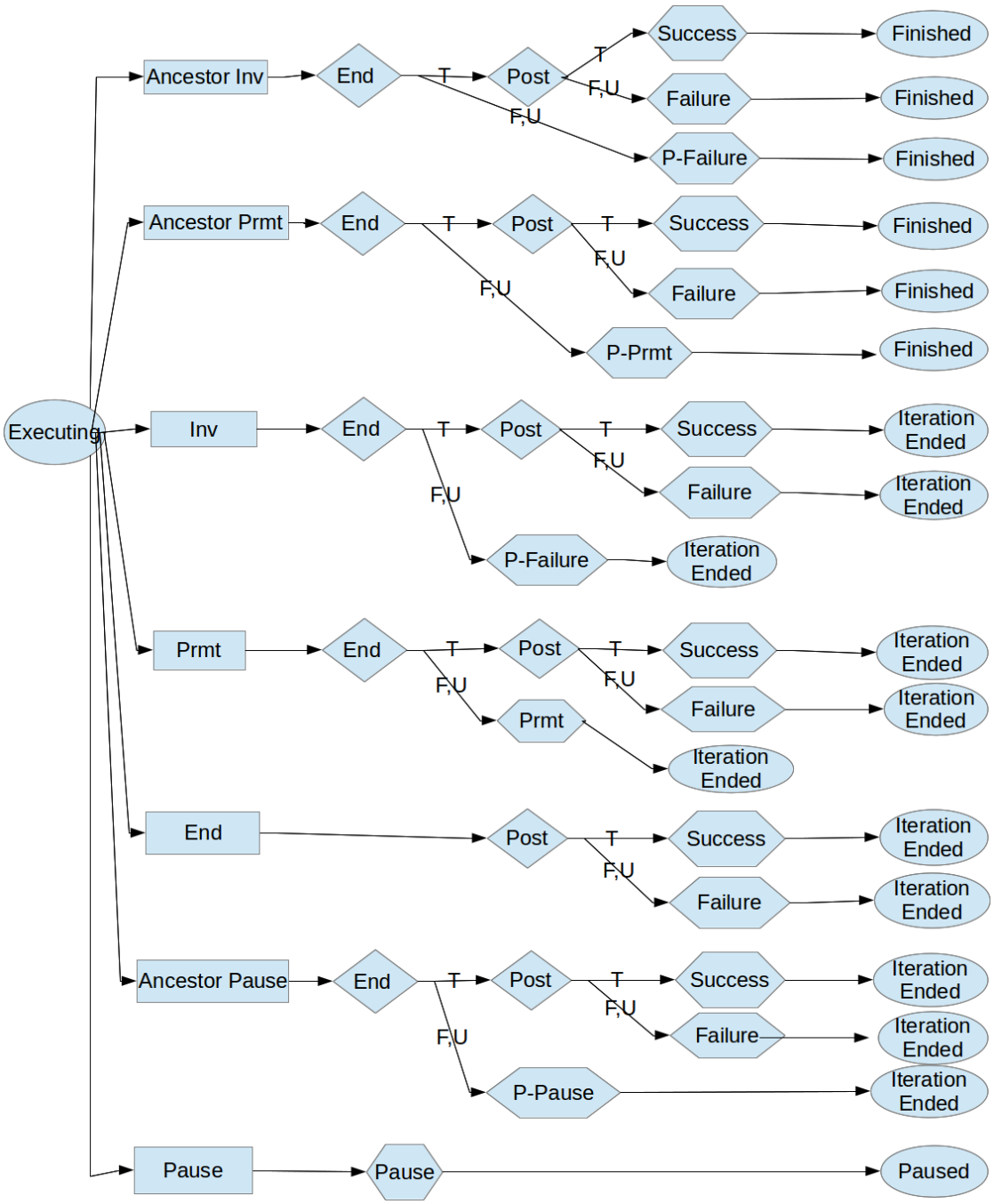


FIGURE B.11: Transitions of child nodes from the Executing state

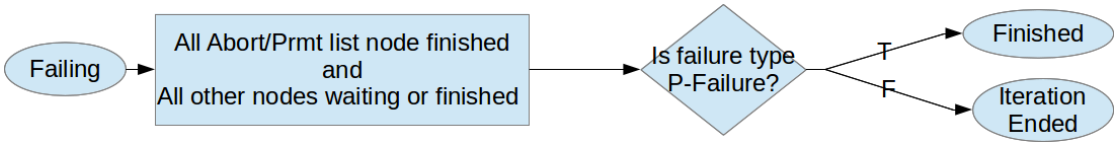


FIGURE B.12: Transitions from the Failing state

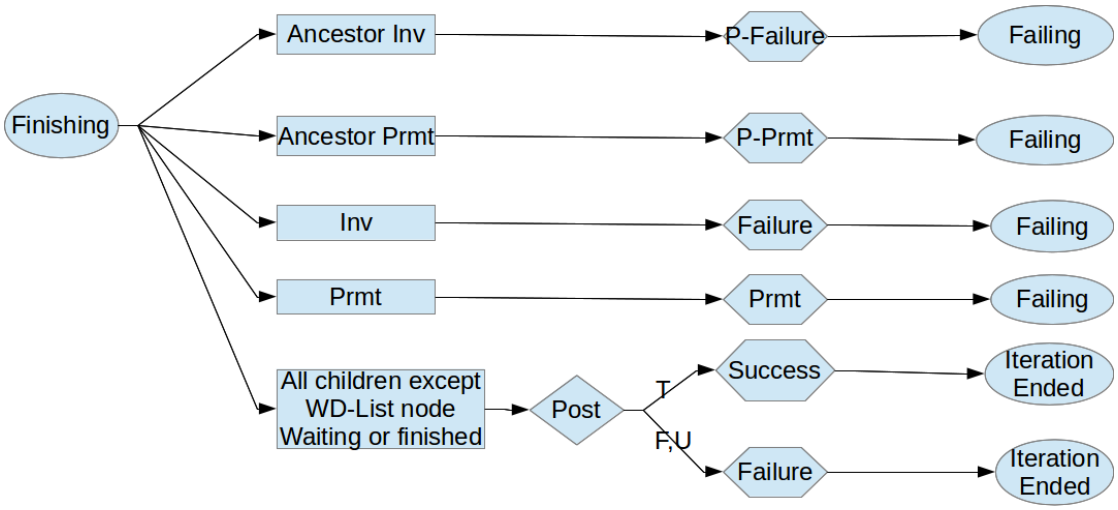


FIGURE B.13: Transitions from the Finishing state

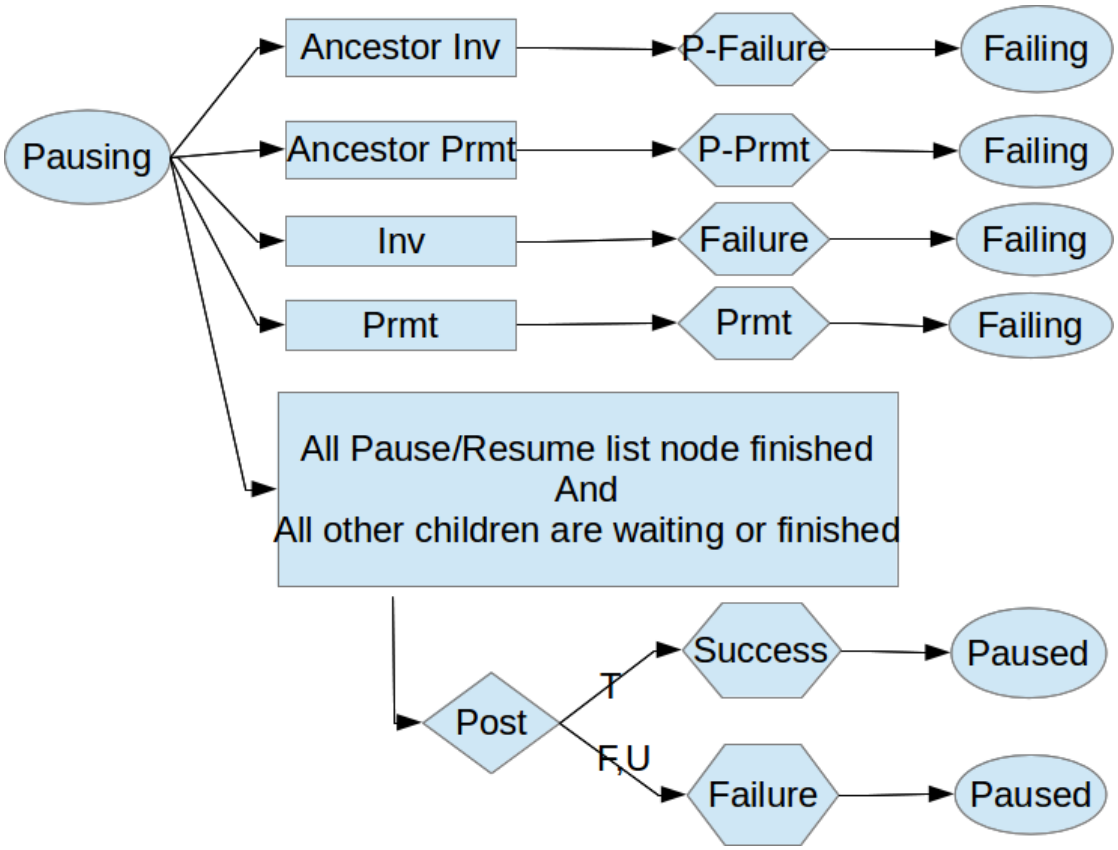


FIGURE B.14: Transitions from the Pausing state

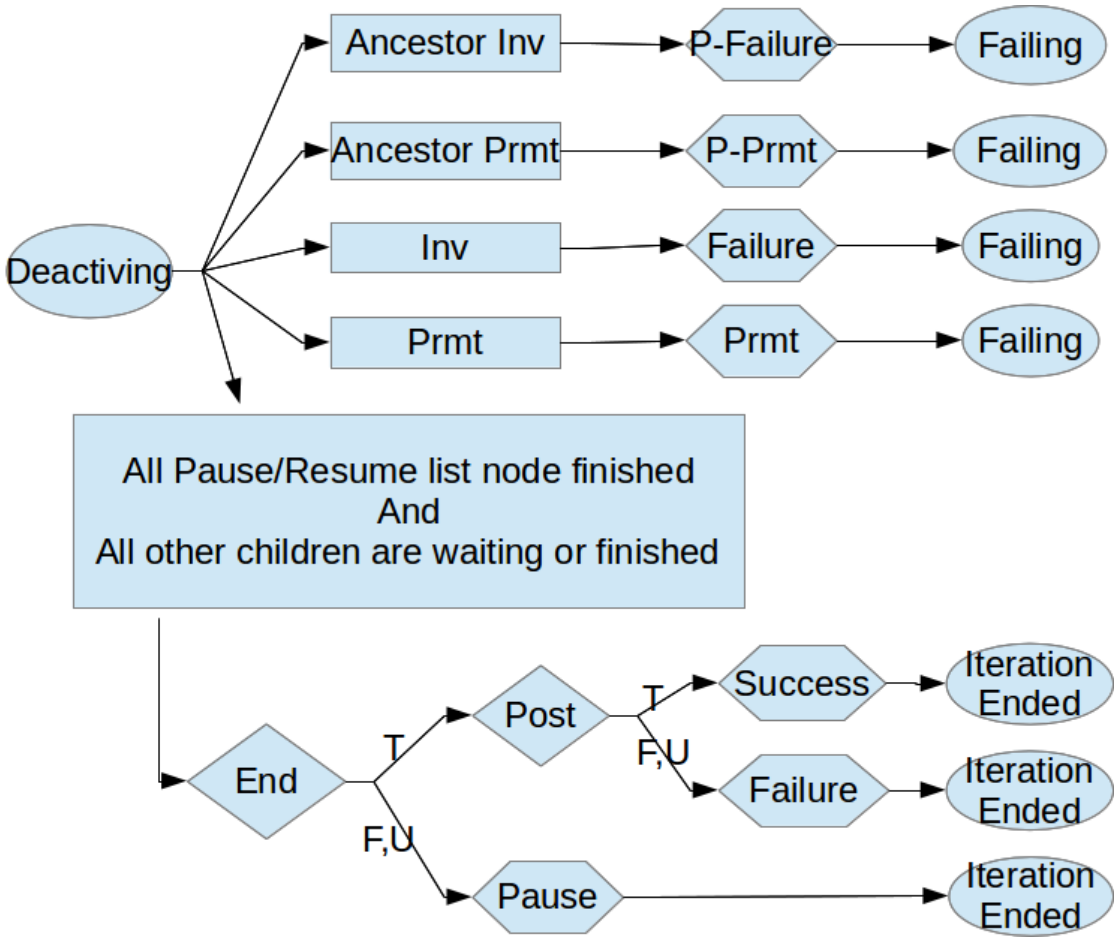


FIGURE B.15: Transitions from the Deactiving state

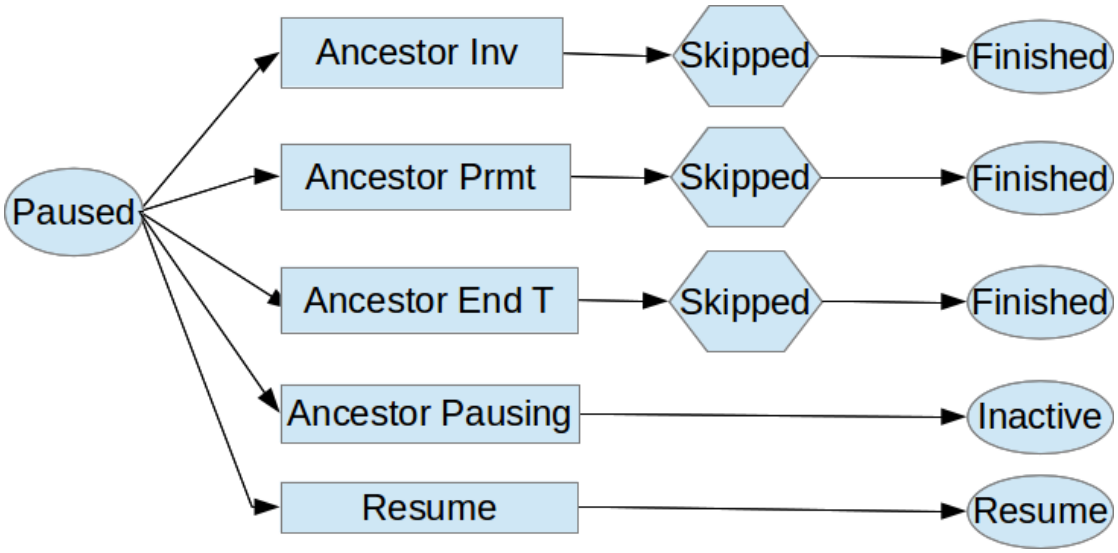


FIGURE B.16: Transitions from the Paused state

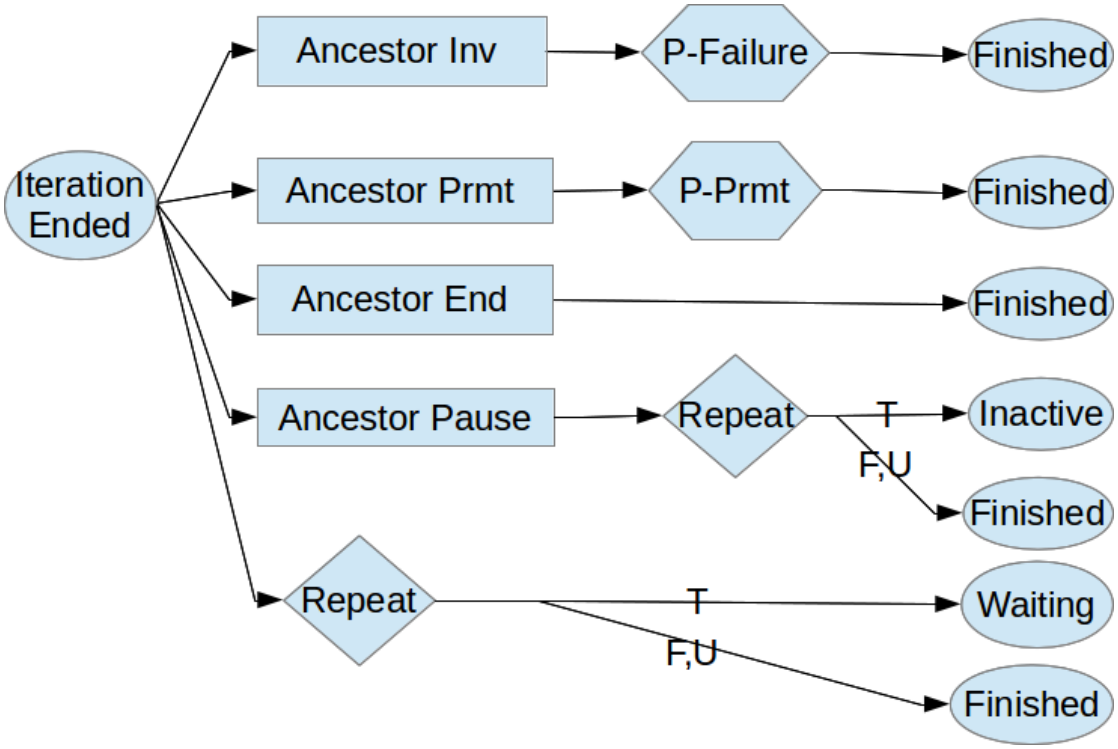


FIGURE B.17: Transitions from the Iteration-Ended state

# Bibliography


Natasha Alechina. Logic and Agent Programming Languages. *Logic, Language, Information, and Computation*, pages 1–10, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-39992-3_1.

Natasha Alechina, Tristan Behrens, Koen V. Hindriks, and Brian Logan. Query caching in agent programming languages. In Mehdi Dastani, Brian Logan, and Jomi F. Hubner, editors, *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS 2012)*, pages 117–131, Valencia, Spain, 06/2012 2012.

Natasha Alechina, Tristan Behrens, Mehdi Dastani, Koen Hindriks, Koen Hubner, Fred Jomi, Brian Logan, Hai H. Nguyen, and Marc van Zee. Multi-cycle query caching in agent programming. In *Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI-13)*, July 2013. URL publications/alechina2013.pdf.

James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983. ISSN 00010782. doi: 10.1145/182.358434. URL http://dl.acm.org/citation.cfm?id=182.358434.

Darko Anicic. Event Processing and Stream Reasoning with ETALIS. *PhD Thesis, Karlsruher Institute of Technology*, 2011. URL http://d-nb.info/1019790091/34.

Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems SE - 5*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15917-6. doi: 10.1007/978-3-642-15918-3_5. URL http://dx.doi.org/10.1007/978-3-642-15918-3_5.

Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963495. URL http://doi.acm.org/10.1145/1963405.1963495.

Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Real-time complex event recognition and reasoning–a logic programming approach. *Applied Artificial Intelligence*, 26(1-2):6–57, 2012. doi: 10.1080/08839514.2012.636616. URL http://www.tandfonline.com/doi/abs/10.1080/08839514.2012.636616.

Krzysztof R Apt and M H van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, 29(3):841–862, July 1982. ISSN 0004-5411. doi: 10.1145/322326.322339. URL http://doi.acm.org/10.1145/322326.322339.

Alexander Artikis, Georgios Paliouras, François Portet, and Anastasios Skarlatidis. Logic-based representation, reasoning and machine learning for event recognition. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems - DEBS '10*, page 282, New York, New York, USA, 2010. ACM Press. ISBN 9781605589275. doi: 10.1145/1827418.1827471. URL http://portal.acm.org/citation.cfm?doid=1827418.1827471.

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

Franz Baader, Ian Horrocks, and Ulrike Sattler. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier, 2008. ISBN 9780444522115. doi: 10.1016/S1574-6526(07)03003-9. URL http://www.sciencedirect.com/science/article/pii/S1574652607030039.

Yaakov Bar-Shalom and Thomas E. Fortmann. *Tracking and Data Association*. Academic Press Professional, Inc, 1988. ISBN 978-0120797608. URL http://books.google.lu/books/about/Tracking_and_Data_Association.html?id=B_FQAAAAMAAJ&pgis=1.

Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Rec.*, 39(1):20–26, September 2010. ISSN 0163-5808. doi: 10.1145/1860702.1860705. URL http://doi.acm.org/10.1145/1860702.1860705.

C. Bauckhage, S. Wachsmuth, M. Hanheide, S. Wrede, G. Sagerer, G. Heidemann, and H. Ritter. The visual active memory perspective on integrated recognition systems. *Image and Vision Computing*, 26(1):5–14, January 2008. ISSN 02628856. doi: 10.1016/j.imavis.2005.08.008. URL http://www.sciencedirect.com/science/article/pii/S0262885606000643.

Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3–4):255 – 299, 1991. ISSN 0743-1066. doi: http://dx.doi.org/

10.1016/0743-1066(91)90038-Q. URL http://www.sciencedirect.com/science/article/pii/074310669190038Q. Special Issue: Database Logic Progamming.

Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM — A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017. IEEE, October 2010. ISBN 978-1-4244-6674-0.

Mark Birbeck. *Professional XML*. Wrox Press, 2001.

Nico Blodow, Dominik Jain, Zoltan-Csaba Marton, and Michael Beetz. Perception and probabilistic anchoring for dynamic world state logging. *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pages 160–166, December 2010. doi: 10.1109/ICHR.2010.5686341.

Mario Bollini, Jennifer Barry, and Daniela Rus. Bakebot: Baking cookies with the pr2. *The PR2 Workshop: Results, Challenges and Lessons Learned in Advancing Robots with a Common Platform, IROS*, 2011. URL http://web.mit.edu/mbollini/Public/icra/BakebotICRASubmit.pdf.

Rafael H. Bordini and Jomi F. Hübner. Bdi agent programming in agentspeak using jason. In *IN: PROCEEDINGS OF 6TH INTERNATIONAL WORKSHOP ON COMPUTATIONAL LOGIC IN MULTI-AGENT SYSTEMS (CLIMA VI). VOLUME 3900 OF LNCS*, pages 143–164. Springer, 2005.

Rafael H Bordini, Lars Braubach, Jorge J Gomez-sanz, Gregory O Hare, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *The Slovene Society Informatika, Ljubljana, Slovenia*, pages 1–15, 2006. URL http://www.informatica.si/PDF/30-1/02_Bordini-ASurveyofProgrammingLanguagesandPlatforms...pdf.

Michael E Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999. ISBN 1575861925. URL http://www.amazon.co.uk/exec/obidos/ASIN/1575861925/citeulike00-21.

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

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

Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

J Buford, G Jakobson, and L Lewis. Extending BDI multi-agent systems with situation management. *The Ninth International Conference on Information Fusion, Florence, Italy*, 2006.

K. Selçuk Candan, Huan Liu, and Reshma Suvarna. Resource description framework: Metadata and its applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001. ISSN 1931-0145. doi: 10.1145/507533.507536. URL http://doi.acm.org/10.1145/507533.507536.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. Logic programming and databases. 1990.

Weidong Chen and David S Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)*, 43(1):20–74, 1996.

L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12(3):359–382, August 1996. ISSN 0824-7935. doi: 10.1111/j.1467-8640.1996.tb00267.x. URL http://doi.wiley.com/10.1111/j.1467-8640.1996.tb00267.x.

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

Silvia Coradeschi and Alessandro Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, May 2003. ISSN 09218890. doi: 10.1016/S0921-8890(03)00021-6. URL http://linkinghub.elsevier.com/retrieve/pii/S0921889003000216.

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

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

Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, March 2008. ISSN 1387-2532. doi: 10.1007/s10458-008-9036-y. URL http://www.springerlink.com/index/10.1007/s10458-008-9036-y.

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

Rosen Diankov and James Kuffner. Openrave: A planning architecture for autonomous robotics. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, 79, 2008.

Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, February 2009. ISSN 1387-2532. doi: 10.1007/s10458-009-9079-8. URL http://link.springer.com/10.1007/s10458-009-9079-8.

Patrick Doherty, Fredrik Heintz, and Jonas Kvarnström. Robotics, Temporal Logic and Stream Reasoning. *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-19)*, pages 42–51, 2014. URL http://www.diva-portal.org/smash/record.jsf?pid=diva2:664665.

Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46–51. IEEE, 2011.

J. Elfring, S. van den Dries, M.J.G. van de Molengraft, and M. Steinbuch. Semantic world modeling using probabilistic multiple hypothesis anchoring. *Robotics and Autonomous Systems*, 61(2):95–105, December 2012. ISSN 09218890. doi: 10.1016/j.robot.2012.11.005. URL http://linkinghub.elsevier.com/retrieve/pii/S0921889012002163.

Orphen Etzion. Event processing and the babylon tower. *Event Processing Thinking blog: http://epthinking.blogspot.com/2007/09/event-processing-and-babylon-tower.html.*, 2007.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003. ISSN 03600300. doi: 10.1145/857076.857078. URL http://dl.acm.org/citation.cfm?id=857076.857078.

Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013. doi: 10.1109/TePRA.2013.6556373.

Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(82)90020-0. URL http://www.sciencedirect.com/science/article/pii/0004370282900200.

Michael Freed. Managing Multiple Tasks in Complex, Dynamic Environments . *In Proceeding of the I998 National Conference on Artificial Intelligence, Madison, WI*, 1998.

Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In Kenneth D. Forbus and Howard E. Shrobe, editors, *AAAI*, pages 677–682. Morgan Kaufmann, 1987. URL http://dblp.uni-trier.de/db/conf/aaai/aaai87.html#GeorgeffL87.

Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.

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

Gilles Dowek, César Muñoz and Corina Pasareanu. A small-step semantics of PLEXIL. *NIA Technical Report*, August 2010. ISSN 01676423. URL http://linkinghub.elsevier.com/retrieve/pii/S0167642313001652.

Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Record*, volume 22, pages 157–166. ACM, 1993.

Christian Halashek-Wiener, Bijan Parsia, and Evren Sirin. Description Logic Reasoning with Syntactic Updates. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-48287-1. doi: 10.1007/11914853. URL http://www.springerlink.com/index/10.1007/11914853.

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

Nick Hawes and Marc Hanheide. CAST: Middleware for memory-based architectures. *Proceedings of the AAAI Robot Workshop: Enabling Intelligence Through Middlware*, 2010. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.174.6622&rep=rep1&type=pdf.

Nick Hawes and Jeremy Wyatt. Engineering intelligent information-processing systems with CAST. *Advanced Engineering Informatics*, 24(1):27–39, 2010. URL http://www.sciencedirect.com/science/article/pii/S1474034609000469.

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

F Heintz. Semantically grounded stream reasoning integrated with ROS. *Intelligent Robots and Systems (IROS)*, 2013. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6697217.

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

Fredrik Heintz and D De Leng. Semantic information integration with transformations for stream reasoning. *International Conference on Information Fusion (FUSION 2013)*, 2013. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6641314.

Fredrik Heintz, Jonas Kvarnstrom, and Patrick Doherty. A stream-based hierarchical anchoring framework. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5254–5260. IEEE, October 2009. ISBN 978-1-4244-3803-7. doi: 10.1109/IROS.2009.5354372. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5354372http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5354372.

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

Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Bridging the sense-reasoning gap: DyKnow – Stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, January 2010b. ISSN 14740346. doi: 10.1016/j.aei.2009.08.007. URL http://linkinghub.elsevier.com/retrieve/pii/S1474034609000524.

Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Stream-Based Hierarchical Anchoring. *KI - Künstliche Intelligenz*, 27(2):119–128, March 2013. ISSN 0933-1875. doi: 10.1007/s13218-013-0239-2. URL http://link.springer.com/10.1007/s13218-013-0239-2.

Koen Hindriks. *Programming Rational Agents in GOAL*, pages 119–157. Springer US, 2009. ISBN 978-0-387-89298-6. doi: 10.1007/978-0-387-89299-3_4. URL http://dx.doi.org/10.1007/978-0-387-89299-3_4.

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

Dominik Jain, Lorenz Mosenlechner, and Michael Beetz. Equipping robot control programs with first-order probabilistic reasoning capabilities. In *2009 IEEE International Conference on Robotics and Automation*, pages 3626–3631. IEEE, May 2009. ISBN 978-1-4244-2788-8. doi: 10.1109/ROBOT.2009.5152676. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5152676.

Ben Kehoe, Akihiro Matsukawa, Sal Candido, James Kuffner, and Ken Goldberg. Cloud-based robot grasping with the google object recognition engine. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4263–4270. IEEE, 2013.

Fernando Koch. An agent-based model for the development of intelligent mobileservices. *PhD thesis, Utrecht University*, 2009.

Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

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

Pat Langley, John E. Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, June 2009. ISSN 13890417. doi: 10.1016/j.cogsys.2006.07.004. URL http://dl.acm.org/citation.cfm?id=2298643.2298716.

Séverin Lemaignan. Grounding the Interaction: Knowledge Management for Interactive Robots. *PhD Thesis, Laboratoire d'Analyse et d'Architecture des Systèmes (CNRS) - Technische Universität München*, 2012. URL http://d-nb.info/1030099995/34.

Séverin Lemaignan, Raquel Ros, E. Akin Sisbot, Rachid Alami, and Michael Beetz. Grounding the Interaction: Anchoring Situated Discourse in Everyday Human-Robot Interaction. *International Journal of Social Robotics*, 4(2):181–199, November 2011. ISSN 1875-4791. doi: 10.1007/s12369-011-0123-x. URL http://www.springerlink.com/index/10.1007/s12369-011-0123-x.

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

Gi Hyun Lim, Il Hong Suh, and Hyowon Suh. Ontology-Based Unified Robot Knowledge for Service Robots in Indoor Environments. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 41(3):492–509, May 2011. ISSN 1083-4427. doi: 10.1109/TSMCA.2010.2076404. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5605259.

J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc. New York, NY, USA, November 1984a. ISBN 0-387-13299-6. URL http://dl.acm.org/citation.cfm?id=2214.

John Wylie Lloyd. *Foundations of logic programming*. Springer-Verlag, 1984b. ISBN 3540132996. URL http://books.google.lu/books/about/Foundations_of_logic_programming.html?id=8uYmAAAAMAAJ&pgis=1.

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

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

Nikolaos Mavridis and Deb Roy. Grounded Situation Models for Robots: Where words and percepts meet. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4690–4697. IEEE, October 2006. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4059158.

Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):43–48, 2006. ISSN 1729-8806.

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

Karen L. Myers. A procedural knowledge approach to task-level control. In *In Proceedings of the Third International Conference on AI Planning Systems*. AAAI Press, 1996.

Nils J. Nilsson. Shakey the robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984.

Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.

Federico Pecora, Marcello Cirillo, Francesca Dell Osa, Jonas Ullberg, and Alessandro Saffiotti. A constraint-based approach for proactive, context-aware human support. *Journal of Ambient Intelligence and Smart Environments*, 4:347–367, 2012. doi: 10.3233/AIS-2012-0157. URL http://iospress.metapress.com/index/L3263N115817W5T6.pdf.

Christian Peters, Thomas Hermann, and Sven Wachsmuth. User Behavior Recognition For An Automatic Prompting System - A Structured Approach based on Task Analysis. *Proceedings of the 1st Int. Conf. on Pattern Recognition Applications and Methods (ICPRAM)*, 2:171, 2012. URL http://pub.uni-bielefeld.de/publication/2423452.

R Píbil, P Novák, Cyril Brom, and Jakub Gemrot. Notes on pragmatic agent-programming with Jason. *Programming Multi-Agent Systems*, pages 58–73, 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-31915-0_4.

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

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

IV Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S Warren. Efficient access mechanisms for tabled logic programs. *The Journal of Logic Programming*, 38(1):31–54, 1999.

Surangika Ranathunga, Stephen Cranefield, and Martin Purvis. Identifying Events Taking Place in Second Life Virtual Environments. *Applied Artificial Intelligence*, 26(1-2): 137–181, January 2012. ISSN 0883-9514. doi: 10.1080/08839514.2012.629559. URL http://www.tandfonline.com/doi/abs/10.1080/08839514.2012.629559.

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

Anand S Rao and Michael P Georgeff. Modeling Rational Agents within a BDI-Architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.5675.

Ricardo Rocha, Cláudio Silva, and Ricardo Lopes. On applying program transformation to implement suspension-based tabling in prolog. In *Logic Programming*, pages 444–445. Springer, 2007.

Robert J Ross. MARC-Applying Multi-Agent Systems to Service Robot Control. *Master Thesis, University College Dublin*, 2003.

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

L. Sabri, A. Chibani, Y. Amirat, and G. P. Zarri. Narrative reasoning for cognitive ubiquitous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, 2011. URL http://hal.archives-ouvertes.fr/hal-00920155.

Diptikalyan Saha. Incremental Evaluation of Tabled Logic Programs. *PhD Thesis, Department of Computer Science - Stony Brook University*, 2006.

Diptikalyan Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *In ICLP, volume 2916 of LNCS*, pages 389–406, 2003.

Diptikalyan Saha and CR Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In *Logic Programming*, pages 235–249. Springer, 2005.

Diptikalyan Saha and CR Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *Practical Aspects of Declarative Languages*, pages 215–229. Springer, 2006a.

Diptikalyan Saha and CR Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In *Logic Programming*, pages 56–71. Springer, 2006b.

Daniel R Schlegel and Stuart C Shapiro. The 'ah ha!'moment: When possible, answering the currently unanswerable using focused reasoning. In *In Proceedings of the 36th Annual Conference of the Cognitive Science Society. Vol. 6.*, 2004.

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

Reid Simmons and David Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937. IEEE, 1998.

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007. doi: 10.1016/j.websem.2007.03.004. URL http://dx.doi.org/10.1016/j.websem.2007.03.004.

Yale Song, David Demirdjian, and Randall Davis. Continuous body and hand gesture recognition for natural human-computer interaction. *ACM Transactions on Interactive Intelligent Systems*, 2(1):1–28, March 2012. ISSN 21606455. doi: 10.1145/2133366.2133371. URL http://dl.acm.org/citation.cfm?id=2133366.2133371.

Dennis Stachowicz and Geert-Jan M Kruijff. Episodic-Like Memory for Cognitive Robots. *IEEE Transactions on Autonomous Mental Development*, 4(1):1–16, March 2012. ISSN 1943-0604. doi: 10.1109/TAMD.2011.2159004.

A. Steck and C. Schlegel. Smarttcl: An execution language for conditional reactive task execution in a three layer architecture for service robots. In *Proceedings of the Workshop on DYnamic languages for RObotic and Sensors systems (DYROS/SIMPAR)*, Germany, 2010.

Terrance Swift and David Scott Warren. XSB: extending prolog with tabled logic programming. *CoRR*, abs/1012.5123, 2010. URL http://arxiv.org/abs/1012.5123.

J. Corina P. Reid S. Kam T; Tara, E. Ari and V Vandi. Plan Execution Interchange Language (PLEXIL). *NASA Technical Memorandum (TM-2006-213483)*, 2006.

Mori Tenorth and Michael Beetz. KNOWROB — knowledge processing for autonomous personal robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4261–4266. IEEE, October 2009. ISBN 978-1-4244-3803-7. doi: 10.1109/IROS.2009.5354602. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5354602.

Moritz Tenorth. *Knowledge Processing for Autonomous Robots*. PhD thesis, Technische Universität München, 2011.

Moritz Tenorth and Michael Beetz. Knowledge Processing for Autonomous Robot Control. *Proceedings of the AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI. Stanford, CA: AAAI Press, 2012*, 2012. URL http://www.aaai.org/ocs/index.php/SSS/SSS12/paper/download/4258/4620.

Moritz Tenorth, Alexander Clifford Perzylo, Reinhard Lafrenz, and Michael Beetz. The RoboEarth language: Representing and exchanging knowledge about actions, objects,

and environments. *2012 IEEE International Conference on Robotics and Automation*, (3):1284–1289, May 2012. doi: 10.1109/ICRA.2012.6224812.

André Ückermann, Robert Haschke, and Helge Ritter. Real-Time 3D Segmentation of Cluttered Scenes for Robot Grasping. *IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012), Osaka, Japan*, 2012. URL http://pub.uni-bielefeld.de/publication/2530701.

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

M. Verbeek. 3APL as programming language for cognitive robots. *Master Thesis, ICS,Utrecht University*, 2002.

V Verma and A Jónsson. Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. *American Institute of Aeronautics and Astronautics Space Conference*, pages 1–19, 2006. URL http://arc.aiaa.org/doi/pdf/10.2514/6.2006-7449.

Vandi Verma and Ari Jónsson. Survey of command execution systems for NASA spacecraft and robots. 2005. URL http://icaps05.uni-ulm.de/documents/ws-proceedings/ws7-allpapers.pdf#page=96.

Vandi Verma, Tara Estlin, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences. *International symposium on artificial intelligence, robotics and automation in space (iSAIRAS)*, 2005(September):5–8, 2005.

Konstantin Vikhorev, Natasha Alechina, and Brian Logan. *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6039 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, September 2010. ISBN 978-3-642-13337-4. doi: 10.1007/978-3-642-13338-1. URL http://dl.acm.org/citation.cfm?id=2175936.2175938.

David E. Watson. Book review: Blackboard Architectures and Applications Edited by V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum (Academic Press). *ACM SIGART Bulletin*, 1(3):19–20, October 1990. ISSN 01635719. doi: 10.1145/101340.1056294. URL http://dl.acm.org/citation.cfm?id=101340.1056294.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.

R. Wood, P. Baxter, and T. Belpaeme. A review of long-term memory in natural and synthetic systems. *Adaptive Behavior*, 20(2):81–103, December 2011. ISSN 1059-7123. doi: 10.1177/1059712311421219. URL http://adb.sagepub.com/cgi/doi/10.1177/1059712311421219.

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

Pouyan Ziafati. Plexil-like plan execution control in agent programming, 2014. URL https://www.aaai.org/ocs/index.php/WS/AAAIW14/paper/view/8810.

Pouyan Ziafati, Fulvio Mastrogiovanni, and Antonio Sgorbissa. Fast Prototyping and Deployment of Context-Aware Smart Outdoor Environments. *2011 Seventh International Conference on Intelligent Environments*, pages 206–213, July 2011. doi: 10.1109/IE.2011.73. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6063387.

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

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

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

# Summary in English

In order to engage and help in our daily life, autonomous robots are to operate in dynamic and unstructured environments and interact with people. As the robot's environment and its behaviour are getting more complex, so are the robot's software and the knowledge that the robot needs to carry out its operations. In collaborating with a human to bake a cake, for instance, the robot needs a large number of components to perceive and manipulate the objects and to communicate and coordinate the task with the human. It also needs a large body of knowledge such as the cooking instruction, the model of objects and common-sense knowledge such as, "eggs are usually found in the fridge." To cope with such complexity, there has been a large body of research on robotic frameworks and robotic knowledge representation and reasoning systems. Robotic frameworks increase the re-usability of the robot's software by supporting its decomposition into separate components and supporting the configuration, composition, communication and coordination of the components. Robotic knowledge representation and reasoning systems provide common language structures and tools to represent, share and integrate pieces of knowledge and to reason about it. However, there is a lack of tools and mechanisms to support aggregating and correlating sensory data to extract knowledge of the robot's environment and to manage, update and query such changing knowledge in an efficient way.

The robot's sensory components continuously and asynchronously process its sensory data into events, discrete pieces of information. Information engineering is the processing, management and querying of sensory events to create and use knowledge of the robot's environment. To be responsive to the situations of the environment, flows of sensory events should be processed on the fly to detect the occurrence of complex events (i.e. on-flow processing). Also, some information should be extracted and maintained in memory to query the state of the environment in the past (i.e. on-demand processing). In addition, planning and plan execution requires the repeated evaluation of the same queries. Doing so efficiently requires an incremental approach to update the results of

these queries when the robot's knowledge base is updated (i.e. incremental query evaluation). The focus of this thesis is on supporting these three models of information processing in autonomous robot software.

This thesis builds on top of recent advances in logic programming to provide a novel architecture for robotic information engineering. It develops the Retalis language for a high-level and efficient implementation of information engineering functionalities. Based on logic programming, Retalis supports rule-based representation and reasoning about knowledge in all three models of information processing. In particular, Retalis addresses the problem of processing discrete and asynchronous flows of sensory data to efficiently extract, represent and manage the robot's knowledge of the state of the environment which is frequently updated through perception and queried for planning and plan execution. We discuss how Retalis can be used to develop a novel agent-based language for autonomous robot programming and present the design specification of such a language. Retalis has been released as a software package for the widely used ROS robotic framework, making it accessible to the robotic community.

# Samenvatting in het Nederlands

Autonome robots die hulp en ondersteuning moeten bieden in het dagelijkse leven opereren in een dynamische en ongestructureerde omgeving en ze moeten kunnen communiceren met mensen. Naarmate het gedrag van de robot en de omgeving waarin de robot opereert complexer worden, wordt ook de software en benodigde kennis die de robot nodig heeft om zijn taken uit te voeren complexer. Om samen te kunnen werken met mensen moet een robot uit een groot aantal componenten bestaan, voor taken zoals waarneming, manipulatie van voorwerpen, communicatie en coördinatie van handelingen met mensen. Voor het uitvoeren van een taak zoals het bakken van een taart moet een robot bijvoorbeeld kennis hebben van de ingrediënten en bereidingswijze maar ook van praktische zaken zoals "eieren vindt men doorgaans in de koelkast." Om met dergelijke complexiteit om te kunnen gaan is er veel onderzoek verricht naar robotische raamwerken en naar kennisrepresentatie en redeneersystemen voor robots. Door middel van decompositie bevorderen robotische raamwerken de herbruikbaarheid van software voor robots. Bovendien ondersteunen robotische raamwerken taken zoals configuratie, compositie, communicatie en coördinatie van componenten. Kennisrepresentatie en redeneersystemen voor robots bieden algemene taalstructuren en hulpmiddelen voor de representatie, uitwisseling en integratie van kennis. Desondanks bestaat er een gebrek aan hulpmiddelen voor het verzamelen en correleren van sensorische gegevens, voor de extractie van kennis over de omgeving waarin de robot opereert, en voor het managen, updaten en queryen van deze kennis op een efficiënte manier.

De sensorische componenten van de robot verwerken continu op asynchrone wijze sensorische gegevens tot discrete stukken informatie (zogenaamde *events*). Information engineering betreft het verwerken, managen en opvragen van deze events met als doel de creatie en aanwending van kennis over de omgeving waarin de robot opereert. Om responsief te zijn op situaties in de omgeving, moet de stroom van sensorische events *on the fly* verwerkt worden, zodat complexe events gedetecteerd kunnen worden (zogenaamde *on-flow processing*). Bovendien moet sommige informatie uit deze gegevensstroom worden geëxtraheerd en opgeslagen in het geheugen, zodat het mogelijk is om informatie over de toestand van de omgeving in het verleden op te kunnen vragen (zogenaamde

on demand-processing). Tot slot moeten, in planning- en uitvoertaken, dezelfde queries vaak herhaaldelijk worden uitgevoerd. Het efficient uitvoeren van deze queries vraagt om een incrementele benadering teneinde de resultaten van deze queries te updaten wanneer de kennisbank van de robot wordt geüpdatet (zogenaamde *incremental query evaluation*). Deze drie modellen voor gegevensverwerking vormen gezamenlijk het information engineering probleem van deze studie.

In dit proefschrift bouwen wij voort op recente ontwikkelingen in de toepassing van logic programming als architectuur voor robotic information engineering. We beschrijven de *Retalis* taal, die we ontwikkeld hebben voor efficiënte high-level implementatie van information engineering functionaliteit. De Retalis taal is gebaseerd op logic programming en ondersteunt rule-based representatie en redenatie voor de drie eerder genoemde modellen voor gegevensverwerking. De Retalis taal maakt het mogelijk om gegevensstromen bestaande uit discrete en asynchrone sensorische gegevens te verwerken. Retalis ondersteunt daarmee de extractie, representatie en het management van kennis over de toestand van de omgeving waarin de robot opereert, met name wanneer deze kennis veelvuldig wordt geüpdatet als gevolg van waarnemingen, en wordt gequeryd ten behoeve van planning en planuitvoer. Verder bespreken we hoe Retalis kan worden gebruikt voor de ontwikkeling van een nieuwe agent-gebaseerde taal voor het programmeren van autonome robots en we presenteren de ontwerpspecificatie van deze taal. Retalis is uitgebracht als een softwarepakket voor het veelgebruikte ROS robotisch raamwerk en is daarmee beschikbaar voor ontwikkelaars in het vakgebied robotica.

# Curriculum Vitae

**Pouyan Ziafati**

**12-08-1984** Born in Hamedan, Iran;

**1998-2002** High School Studies, National Organization for Development of Exceptional Talents, Iran;

**2002-2008** Software Engineering (BSc), Iran University of Science and Technology, Iran;

**2008-2010** European Master in Advanced Robotics (MSc),

- Warsaw University of Technology, Poland,
- University of Genova, Italy;

**2010-2011** Research Assistant, DERI, Ireland;

**2011-2015** PhD Student,

- Intelligent Systems Group, Utrecht University, Netherlands,
- SnT, University of Luxembourg, Luxembourg;

**2015-2016** Research Associate, SnT, University of Luxembourg.