Robotics software engineering

Architecture for Logic Programming with Arrangements of Finite-State Machines

Autor: Alberto Ramírez Regalado

MASTER THESIS/ September 2016

SUPERVISOR
Vladimir Estivill Castro. Department of Technology Information and Communication.
Acknowledgements

My deepest thanks to Dr. Vladimir Estivill Castro for his guidance and teaching and specially convey his energy and passion for Robotics.

Thanks to my family for everything...
Abstract

Reactive systems are by nature systems without rationality. As its name implies, they are characterized by reacting to the environment. Their quick response and simplicity to build them make them the most used systems in robotics. Since the emergence of Logic-Labeled Finite-State Machines, a high-level description of reactive systems is enabled. The versatility we have with this way of representing behaviors allows us to think of reactive agents that can reason and make logical decisions. The questions therefore are, can we use ubiquitous models of reactive behavior, such a state machines in combination with declarative modeling afforded by Prolog? Can we combine Prolog with reactive systems? What sort of architecture should this take in a robotic system in order to integrate reasoning while maintaining a simple structure of control?

In this thesis I describe the integration of logic programs into reactive systems, based on Logic-Labeled Finite-State Machines, demonstrating a way to add rationality to reactive systems. I use Prolog (the most popular language in logic programming) as a logic engine that helps us to create a knowledge base allowing the systems to query the Prolog program and get answers needed to make decisions about their behavior. The communication between the Finite-State Machines and Prolog is performed by a middleware under the Pull-approach, achieving deterministic semantics and the ability to ensure correctness in both the time and value domains. I also contrast the use of a middleware under a Pull-approach versus a middleware under a Push-approach. The differences between these two approaches reflect the conclusions of this thesis.
Contents

List of figures  xii

1 INTRODUCTION  1
   1.1 Context  ........................................ 1
   1.2 Statement of the problem  ........................ 3
   1.3 Objective  ....................................... 4

2 MAIN CONCEPTS OF THE PROPOSAL  5
   2.1 Logic-Labeled Finite-State Machines  ............. 5
      2.1.1 Definition of Logic-Labeled Finite-State Machines  6
      2.1.2 Characteristics of LLFSM  ........................ 9
      2.1.3 Modeling behavior with LLFSM  ................. 10
   2.2 Middleware  ...................................... 13
      2.2.1 Push and Pull approaches  .................... 14
   2.3 Logic Programming  ................................ 16

3 INCORPORATING LOGICAL THINKING INTO LLFSM  19
   3.1 The selection of Prolog as a logic programming language  19
      3.1.1 Interfacing LLFSM and GNU-Prolog  ............ 20
   3.2 LLFSM PROLOG architecture approach  ............. 21
      3.2.1 Example under the Push approach  ............. 25
4 CASE STUDY
  4.1 Traffic light control problem description .................................. 29
  4.2 Modeling traffic light problem ...................................................... 30
  4.3 LLFSMs implementation ............................................................... 32
  4.4 C_Prolog_wrapper implementation ............................................... 35
  4.5 Sequence execution ..................................................................... 37

5 CONTRASTING THE PULL APPROACH WITH THE PUSH APPROACH .... 41
  5.1 LLFSMs with ROS-services ............................................................ 43
  5.2 Execution sequence ..................................................................... 45
  5.3 Case study contrast between Pull and Push approaches ................ 47

6 FUTURE WORK ................................................................................. 49
  6.1 Poker player and CLUEDO game player ......................................... 49
  6.2 Ontologies knowledge base with Prolog-LLFSM .......................... 50

7 CONCLUSIONS .................................................................................. 51

Appendices ......................................................................................... 53

A C_PROLOG_WRAPPER ................................................................. 55
  A.1 Creating a C_Prolog_wrapper ..................................................... 55
  A.2 Create a Gusimplewhiteboard writer/reader program in C++ ... 58
  A.3 compilation using ROS Catkin .................................................. 60

B GUSIMPLEWHITEBOARD CONTROL/STATUS MESSAGES .......... 61
  B.1 Definition of control/status messages in Gusimplewhiteboard ... 61
C  LLFSM IMPLEMENTATION EXAMPLE 65

C.1 Building behavior with MeditLLFSM editor . . . . . . . . . . . 65
List of Figures

2.1 Basic elements of LLFSM notation. ........................................ 7
2.2 Sequential pseudocode of activities between two states. .......... 8
2.3 A subsumption architecture example. ................................. 10
2.4 CONTROLLER LLFSM of a exploratory behavior depicted in
subsumption architecture. ....................................................... 12
2.5 Basic ROS-service interaction under the Push approach. ......... 15
2.6 Basic Gusimplewhiteboard interaction under Pull approach. ... 16

3.1 A generic model of how a logic engine can be integrated into re-
active systems. ................................................................. 22
3.2 Sequence diagram showing the execution of an on/off light switch
under the Pull approach. ..................................................... 23
3.3 Proposed architecture under the Pull approach. .................... 24
3.4 Proposed architecture under the Push approach. ................... 25
3.5 Sequence diagram showing the execution of an on/off light switch
under the Push approach. ..................................................... 26

4.1 A traffic light control at the intersection of two roads. .......... 29
4.2 Generic model of a traffic light behavior. ............................ 30
4.3 A block diagram composed of 7 modules of the traffic light im-
plementation. ................................................................. 31
4.4 EW LIGHT TRAFFIC CONTROL LLFSM. ............................ 33
4.5 NS LIGHT TRAFFIC CONTROL LLFSM. ............................ 34
4.6 TIMER LLFSM represents a timer behavior. ......................... 35
4.7 Prolog program content of a traffic light control. ................... 36
4.8 Sequence execution diagram of the traffic light control problem solution under the Pull approach .......................... 38

5.1 Block diagram of the traffic lights control problem under the Push approach using Robot Operating System ROS. ............... 42
5.2 The content of the ROS-service file shallMoveNsAmber.srv. .... 43
5.3 Transition from GREEN_ON_NS to AMBER_ON_NS by a ROS-service. ......................................................... 44
5.4 Execution sequence diagram of the solution under the Push approach using Robot Operating System. ......................... 45

A.1 Wrapper functions. ................................................... 58

B.1 Messages in the gusimplewhiteboardtypelist.tsl file. ............. 63

C.1 Definitions of the state Red_On_ENTRY using the LLFSM editor MieditLLFSM. This is part of the traffic light control LLFSM on the East-West side. ..................................................... 66
C.2 The States tab shows different states of the LLFSM. .............. 66
C.3 The Variables tab shows different variables of the LLFSM. ...... 67
C.4 The Transitions tab in the MiEditLLFSM. In here, transitions are defined from state to state. ................................. 67
Chapter 1

INTRODUCTION

1.1 Context

In the modern days, different knowledge areas such as electrical and mechanical engineering, computer science, physics, calculus and others, have merged together and developed new interesting areas of study such as robotics. This fast-growing field is in charge of design, construction, and operation of autonomous systems, like robots.

One of the most accepted definitions of a robot is that “a robot is an autonomous system which exists in the physical world, can sense its environment and can act on it to achieve goals” [21]. In other words, I can say, a robot is a mechanism that acts accordingly to its own decisions, not controlled by a human, with capabilities of sensing the environment through sensors and achieving goals through actuators.

This simple definition is the base of study of robotics and involves different concepts from different fields of study. Control theory is an important concept necessary to understand robotics in general. Control theory is included in almost all mechanical systems we know today. It is fundamental to understand this theory,
in order to design and build machines. The governability of a machine depends almost entirely on its control. Typical problem examples are temperature control, water systems control, and power control. The solutions are applied to a variety of common machines like microwave ovens, airplanes, cars, air conditioning, robots and others. The key concept of control theory focuses on coupling, combining, and interacting with the mechanism and its environment.

Current approaches to robot control emerged fundamentally from the influences of and lessons learned from control theory (the mathematics of controlling machines) [21, 1]. M.J. Mataric summarizes the robot control problem in this phrase: “The complexity of the robot control problem is directly related to how unpredictable and unstable the environment is, to how quickly the robot must react to it”. In addition, Mataric classifies the types of robot control as follows[21]:

1. Reactive Control: Do not think, (re)act.

2. Deliberative Control (planner based/ logic based): Think hard, act later.

3. Behavior-Based Control: Think the way you act.

4. Hybrid Control: Think and act independently, in parallel.

One of the most studied architecture of machines and autonomous robots included in the family of reactive architectures for robotics systems is feedback-loop control[1]. In robotic terminology, feedback-loop control enables a robot or embedded system to reach or maintain a desired state (represented by state variables). The system obtains input information (or measurements) through sensors, and the controller produces, as output, a control signal that can affect the state of the system. Particularly, the feedback-loop control is a generic way of using control theory and helps significantly to build reactive systems. Feedback-loop control as part of reactive architecture robots is the most common system since it is the
easiest agent to make. The response time is short and reflective to the surrounding environment changes. Feedback-loop control does not perform symbolic reasoning.

Deliberative control exists on the opposite end of reactive control architecture. It is distinguished for creating systems used to solve difficult problems of artificial intelligence, such as playing chess, where thinking hard is exactly the right thing to do. This kind of control has brought the possibility to build intelligent machines giving certain kind of deliberation to mechanical machines, through computers, algorithms, hierarchical system organization, symbolic representation, planning and reasoning in order to solve problems.

It is also known that deliberation requires a big amount of time for decision processing. Sometimes this amount of time can be critical. Thus, the idea of reducing time in the decision-making process but also to maintain accuracy always brings new proposals. From our point of view, reactive control can be combined with deliberative control in a certain way that a robot or an autonomous mechanism can reduce response time in making decisions and be accurate at the same time. The proposal presented in this thesis is focused on incorporate deliberative/logic based reasoning into reactive systems under the Pull approach.

In the following sections, I will show the particularities of this proposed architecture, and the necessary elements to implement it.

1.2 Statement of the problem

The general problem relates to the safe operation of robots. The general question is: How to represent information about the world in such a way that agents can reason and act with it in an acceptable fixed time-bound?

This work is based on Logic-Labeled Finite-State Machines (LLFSM) repre-
senting behaviors of reactive systems and the possibility to add some deliberation using high-level, declarative specification through logic programming with Prolog. Although the architecture presented does not depend on this particular format of reasoning.

Two important questions arise from this problem:

1. Can be the selection of how to act (reactive architecture) be more reasoned, and the computed model be as transparent as an LLFSM?

2. What sort of architecture should this take in a robotic system in order to integrate reasoning while maintaining a simple structure of control?

These questions are particularly important, as the answer will give us a clear picture of what kind of reactive systems would arise from this paradigm.

1.3 Objective

The general aim of this project is to generate a robot control architecture that responds fast and efficient to the changes of a dynamical environment. The specific objectives of this master thesis are:

1. To make reactive systems perform with logical thinking.

2. To integrate logical thinking (Prolog) into an LLFSM.

3. To contrast the Pull approach with the Push approach.
Chapter 2

MAIN CONCEPTS OF THE PROPOSAL

This chapter will cover the essential concepts of this proposal. Basically, there are four concepts that must be studied in order to understand the solution given to the problem stated in Chapter 1. As we want to make reactive systems acting with logical thinking, the first element I would like to introduce is Logic-Labeled Finite-State Machines which represents behaviors of reactive systems. The second element is the middleware used in a Pull approach which is the third element. The fourth element is the logical thinking using logic programming.

The next sections will explain each one of the concepts involved in the solution.

2.1 Logic-Labeled Finite-State Machines

Finite-State Machines (FSMs) are common elements in robotics systems. FSMs are a useful mechanism to describe the behavior of a system. Although it is a basic architecture, it represents perfectly modern behaviors and it is the basis of
other control architectures like subsumption architecture or the frequently used behavior-based robot control [4].

A well-known example of behavior-based robots is The Mars Exploration Rovers a project performed by Jet Propulsion Telerobotics Research and Applications Group [16, 27] and applied in the Mars exploration project of NASA. This example of NASA shows that reactive systems represented by FSMs are still ongoing architectures.

Early behavior descriptions of FSMs [8] can be emulated by LLFSMs [13], whose transitions are labeled by logical expressions that are compiled directly from C. According to Estivill-Castro & Hexel [10], LLFSM constitutes an executable model that follows a pre-defined, deterministic schedule that stipulates the thread of execution. Besides, LLFSM offers a very clear and transparent semantics, while offering rich enough construct. A minimal set of constructs allows rapid learning and adoption.

LLFSM are analogous to the now very common UML-model derived from OMT [25, chapter 5] and Statemate [14]. They are similar in its composition of states and transitions between states. However, they are radically different in that LLFSM transitions are not labeled by events. What characterizes LLFSMs is that transitions are labeled by logic Boolean expressions. These expressions are equivalent to queries to an inference engine.

### 2.1.1 Definition of Logic-Labeled Finite-State Machines

An LLFSM consist of a set \( S \) of states [8], and a transition function \( T : S \times E \rightarrow S \).

There is a distinguished state \( s_0 \in S \), named the initial state. The set \( E \) is a set of Boolean expressions. This is an important aspect of LLFSM. For example, LLFSMs have used models of logic to express what is the off-side rule in soccer [3], when is dangerous for a senior lady to face a stranger, and to describe hands
of poker [2]. Therefore, logic becomes a very useful tool for communication what criteria classify a soccer player’s position as valid or a set of 5 cards as a full-house.

To explain how an LLFSM works, it is necessary to think in a state diagram. Figure 2.1 shows the basic elements of the notation and Figure 2.2 shows the pseudocode of the activity between these basic elements in Figure 2.1. All transitions out of a given $S$ form a sequence, they have an order. State activity occurs in sections named OnEntry, OnExit, and Internal.

The thread of execution is predefined and sequential. The sequence begins executing actions in the OnEntry section. If the transition expressions become true, then actions in the OnExit section are executed and the machine departs the state. Thus, actions in these two sections are executed only once. Conversely, if the transition expressions become false, then actions in the Internal section are executed. When all actions in the Internal section are completed, execution returns to evaluate the transition expressions and the cycle repeats again. The literature refers to one pass over the cycle as a ringlet [8].

![Figure 2.1: Basic elements of LLFSM notation.](image)

When you are dealing with only one LLFSM, the execution of the ringlet is the task of performing all works associated with the current state. Namely, all transitions leading out from the current state are evaluated in their designed order.
In Figure 2.2 we can observe the sequential program and the notation to describe an arrangement that has only one thread of execution. The consequences of such a sequential model of execution are that activities and actions in different LLFSM within the arrangements do not constitute race conditions and there is no need for
mutual-exclusion mechanisms. Variables across LLFSM can reside in memory without conflict.

Although Figure 2.2 shows clearly sequentiality, it does not reflect the simplicity of LLFSM. LLFSM provides a very brief notation. This notation allows building behaviors in a simple and transparent manner. However, if these behaviors were to be written in a programming language like C, C++ or Java, the code would be extended and complicated with many loops, if and else instructions.

2.1.2 Characteristics of LLFSM

There are some advantages of using LLFSM. As far as I know, this way of representing software and behaviors has not reached its limits of versatility. This thesis is an example of versatility we have with LLFSM because we are adding flexibility of reasoning in addition to removing the fact that they are purely reactive. Although the possibilities and advantages depend on the way you use LLFSM.

The main characteristic of LLFSM is that transitions between states are labeled by Boolean expressions. The values of Boolean expressions in transitions can be determined by a more complex logic than propositional logic provided by C language. In particular, I propose that transitions are determined by the evaluation of a Prolog program.

Another characteristic of LLFSM is deterministic execution time. This execution time corresponds to a defined sequential schedule and has been explained in Figure 2.2. Sequential execution has been proposed as superior over the multiplication of concurrent threads [22]. This approach brings with the benefit of concurrency, facilitating modular, component-base design.
2.1.3 Modeling behavior with LLFSM

Modularity is one of the great advantages of LLFSM. Brooks [4] introduced the subsumption architecture for robotics and automated control systems. The key to this architecture is a layering of control systems to establish complex behavior that can be composed of individual, simpler behaviors by selectively suppressing control systems that implement currently undesired behaviors. This can be seen as atomic behaviors that constitute a large complex behavior. Modules that work together to build large systems.

Figure 2.3: A subsumption architecture example.

Figure 2.3 shows an example of modularity and composition in LLFSM. Also, it shows how the union of independent behaviors can achieve complex behaviors through a machine controller. In the figure mentioned recently, there are three behaviors named Behavior A, Behavior B, and Behavior C. Hierarchically Behavior A is above of Behavior B and Behavior C. It means that machine A is the behavior...
controller. This module contains independent states that control when Behavior B or Behavior C should start or suspend. Behavior B and Behavior C are modules that can act separately and perform their own actions independently. This modularity is achieved through a model of state machine suspension.

In order to demonstrate modularity while implementing the architecture of subsumption, it is necessary to show a practical example. A typical example in robotics is the creation of a mobile robot that can autonomously explore an area of land. This problem can be easily solved with a behavior-based/subsumption architecture. Three LLFSMs work together to achieve the exploratory behavior we need. The general behavior of exploring consists of a master controller and two sub-behaviors with specific activities.

Suppose we have three LLFSMs named CONTROLLER, AVOID, and STROLL. For simplicity, AVOID and STROLL are not shown in a figure but suppose they exist. Let’s assume AVOID is in charge of avoiding collision and STROLL is in charge of the stroll activity. CONTROLLER is the master controller. It has the ability to delegate responsibilities to other machines that are dedicated to specific activities as appropriate to the current scenario.

Figure 2.4 illustrates what happens inside the LLFSM CONTROLLER showing how activities are delegated to other LLFSMs. Three states are shown: DECIDING, RunAvoidSubMachine, and RunStrollSubMachine. Early in the process, the state DECIDING decides to suspend both sub-behaviors (LLFSMs). The way it does uses SUSPEND AVOID and SUSPEND STROLL. After that, the conditions for deciding who receives the token are evaluated. If there is an obstacle in front of the robot, DECIDING sends the token to the state RunAvoidSubMachine. Once there in the OnEntry section, we resume AVOID machine using RESUME AVOID. AVOID performs to achieve the objective of avoiding an obstacle fulfilling output conditions that suspend the LLFSM AVOID and return the token to
DECIDING. Now the robot is free of obstacles so the conditions to pass the token to RunStrollSubMachine become true. Once there, the LLFSM STROLL is resumed. STROLL runs until the robot meets another obstacle. Consequently, the output conditions are valued at true and the LLFSM STROLL is suspended and the token returns to DECIDING. The cycle begins again.

In summary, three expressions of LLFSM language; SUSPEND, RESUME, and RESTART allow many machines to work together to achieve complex behaviors. Simple, but flexible concepts which allow us to model the subsumption architecture programs in the hierarchy of states. This is a powerful feature of LLFSM that makes it a complete language for designing robots, specially reactive and behavior-based controlled.
2.2 Middleware

The need for people to interact easily and better with systems and digital devices has caused that these systems increase their complexity, requiring faster and more sophisticated software. Such software often interacts with various modules of hardware and even other modules of software, increased the complexity of gathering all in one system.

This complexity in systems, the diversity of them, and built by different manufacturers, have inadvertently created the need to engage in one system different subsystems that together reach sophisticated behaviors. Thus, so the concept of modularity is created. Start at the same time the need for something that could integrate all the different modules. This something is software known as middleware.

Middleware can be defined as the name implies, as software that exists amongst various objects or modules. Middleware is an intermediary that allows other modules to interact amongst themselves. In a way, it centralizes communications between modules. That is, instead of modules communicate directly with each other, they communicate via middleware. In the case of robotics, these modules can be software or hardware. Actually what really matter is that the interaction between different parts is achieved through the middleware. Many of the middleware are characterized by the use of blackboard control architecture [15] to integration modules, to helping modules handle behaviors and managing problem-solving tasks [9].

There is a variety of middleware for robots, however along this thesis we focus on two of them. The first one is the blackboard Gusimplewhiteboard, develop by Machine Intelligence and Pattern Analysis Laboratory Mipal [17]. The second one is the Robot Operating System also called ROS [24]. Both middlewares have different characteristics that contrast with each other basically by the approach
they use. On the one hand, Gsimplewhiteboard uses the Pull approach to communicating their modules. On the other hand, ROS uses the Push approach.

2.2.1 Push and Pull approaches

To explain the Pull and Push approaches, it is necessary to describe the process of communication between two modules using a middleware or blackboard. In a basic communication, there are two important elements, the sender, and the receiver. The sender is the one that produces the information and tries to send it. The receiver is receiving the information. The blackboard enables a sender to post the information. This operation is usually a non-blocking call. That is to say, it is as simple as writing a message on the blackboard that serves as a repository. The difference between the Push and Pull approach lies in the way that message is retrieved by the receiver. Based on whether the reception of a message is within the sphere of control (SoC) of the sender or the receiver, there are essentially two modes [9]:

\[ \text{subscribe}(T,f) \]: In this mode, the receiver of the message subscribes to this type of message \( T \). When a module has written a message of type \( T \) on the board, this notifies the recipient that there is a message. This procedure is done by a callback function \( f \). This way of recovering the message is known as the Push technology.

\[ \text{get\_message}(T) \]: In this mode, the recipient retrieves the message from the board by calling the function \( \text{get\_message}(T) \). Then the board provides the receiver the last message of the type \( T \) that has been published. This is what is called Pull technology.

These two modes are very common in software architecture. However, when it comes to robotics, the most popular mode used is the Push approach. This is nothing but a technology that is distinguished by a subscriber with a callback. The
most famous representative of this approach is ROS [24].

ROS provides two mechanisms of communication, ROS-topics, and ROS services. In this thesis, we focus on ROS-services. An ROS-service is basically a remote procedure call. This implies synchronization which is a characteristic of the Push approach. Figure 2.5 depict a typical ROS-service communication. In this communication, the consumer offers a service and a client uses the service by sending a request message through the middleware and awaiting the reply.

![Basic ROS-service interaction under the Push approach.](image)

This communication requires coupling between the parties. It means a delay in obtaining a response. Also, the consumer who makes the call need to fulfill the inputs needed for the callback function. Suppose those inputs do not come from the same node, it means that the producer must collect those inputs from other nodes and therefore need more couplings.

In contrast to ROS and its Push model, there is a blackboard called Gusimplewhiteboard [11] that uses Pull technology to interact amongst its components. This communication is characterized by its components read and write messages at their own pace. These messages are stored on the blackboard ad can be classified as control messages and status messages. Control messages indicate actions desired by the nodes, whereas status messages indicate the current status of any node or module. There are some advantages of using these messages as idempotent, fault recovery is one of them. Estivill-Castro & Hexel [10] refer to idempotent control/status messages as: “What makes them idempotent is the fact that
receiving the same message multiple times has no different effect than receiving that message (correctly) once”.

![Diagram of Gusimplewhiteboard interaction under Pull approach.](image)

Gusimplewhiteboard is a shared memory object-oriented blackboard. Control and status messages on the blackboard are objects of the same class. Actually, this board is just a library that can be included in any node. Figure 2.6 shows a basic Gusimplewhiteboard interaction under the Pull pattern. We can observe how the producer of a message writes the message on the blackboard without knowing who will consume this message. Neither reported any of the nodes that a message has been written. Consumers retrieve this message when it best suits them.

### 2.3 Logic Programming

Logic programming can be analyzed as a controlled deduction. This idea is summed up in Robert Kowalski’s equation [18]:

\[
\text{Algorithm} = \text{Logic} \pm \text{Control}
\]  

(2.1)

Where \text{Logic} represents a logic program which is a statement that describes the problem that has to be solved. \text{control} represents a different theorem-proving strategy which is a statement of how the problem has to be solved. It is clear
that this algorithm is related to the solution of a problem, made specifically for making decisions that involve agent actions, in our case robot actions. Generally speaking, a logic programming system should provide ways for the programmer to specify each of the components of the Kowalski equation 2.1 [19].

According to Russel & Norvig [26], a logical program consists of two elements; the knowledgebase and logical inferences. Logical program consists of two key elements; the knowledge base and queries. On one hand, the knowledgebase is a model of an environment in which facts are represented. In simple words, it is an expert system that we want to ask questions. The knowledgebase is made up of sentences into a declarative language. Each sentence must satisfy the model we want to represent. Proper relationships between these sentences derive in drawing logical inferences. Logical inferences are achieved by performing a search strategy amongst the knowledgebase. This search represents a challenge is this thesis because the time requirements change from search to search, even against the same knowledgebase.

Prolog is the logic programming language most used in the world. It is simple and powerful. It is based on the principles of first-order logic. The next chapter will show some of the Prolog characteristics, and the reason for choosing it as part of this thesis.
Chapter 3

INCORPORATING LOGICAL THINKING INTO LLFSM

3.1 The selection of Prolog as a logic programming language

For the purpose of this thesis, we have chosen Prolog as a logic programming language, in principle because is a well know language with plenty of documentations on textbooks and the web, which makes it a perfect language to test Logic programming with LLFSMs. However the architecture we are proposing in this work does not depend on this particular format of reasoning. It is illustrative of the main scenarios where the logic engine performs some heuristic search, and thus event task oriented reasoning like GOLOG [5] could be incorporated in a similar architecture.

Other feature that makes Prolog a good language for this architecture experiment is that combines procedural interpretation with declarative readings that enables to inspect the content of the knowledge base. Its depth-first search with
backtracking enables the programmer to control search by ordering clauses of queries and rules.

The possibility to write procedures in Prolog speeds up inferences. This feature makes Prolog especially interesting because we can write complicated systems in Prolog without the necessity of another programming language. Although in combination with LLFSM we used C++ as a common executable language. For this particular reason, we have selected GNU Prolog [6] as it offers a direct interface to C and C++.

Another Prolog characteristic is that facilitates external information sources or reasoning engines. This is useful if we want to go a step further on reasoning mechanisms because we can use extensions packages of prolog to connect with OWL or RDF in order to use ontologies and other interfaces to advanced intelligent systems. Although this feature has not been used in this thesis, it opens the possibilities of future works on this research line, like in KNOWROB [29] which has been used as the basis of elaborate robotics systems for the challenge of RoboCup@Home [23].

All these factors made Prolog a strong candidate to experiment with LLFSM, with the firm intention to grant certain logic to reactive systems.

### 3.1.1 Interfacing LLFSM and GNU-Prolog

As we have the intention to create a connection between the Boolean expression transitions of the states and the responses of the inferences of the Prolog program, is necessary to create a wrapper between two programming languages, C and Prolog, and we call it C-Prolog_wrapper.

With GNU Prolog [6] is possible to call (or callback) a Prolog predicate from a C function. The C code can ask for next solutions, remove all remaining solutions or terminate and keep alternatives for the calling Prolog predicate [5]. The header
gprolog.h is given by GNU Prolog and the construction of the wrapper follows simple steps.

### 3.2 LLFSM PROLOG architecture approach

Figure 3.1 shows a generic model of how a logic engine can be integrated into reactive systems represented by LLFSMs. What we want to highlight in this Figure are the necessary components of our approach and the working relationship between the modules.

In Figure 3.1 Module Behavior represents reactive systems through LLFSMs. In Section 2.1.3 we have described how an LLFSM or a set of them can build both simple and complex behaviors.

In Figure 3.1 Module Thinker represents a logic engine and its knowledge base. Specifically, Prolog allows queries to a knowledge base and sends the results to Module Behavior through the middleware.

In Figure 3.1 Module Sensors & Actuators is the representation of how a robotic system interacts with the physical environment. For the purpose of this thesis is irrelevant, however, it helps us understand a complete robotic system using this architecture.

Finally, in Figure 3.1 Module Middleware represents the middleware.

To start with the details of our proposal is necessary to mention the three crucial elements of this architecture:

1. The first element comprises the use of Logic-Labeled Finite-State Machines (LLFSM).

2. The second element is that the communication between the LLFSM and thus, the reactive system and the reasoning engine, is performed under the Pull approach rather than the Push approach.
3. As the third element, we argue for using control/status messages to define communication channels that minimize the need for explicit delivery guarantees by the communication channel or middleware or the need for explicit acknowledgments. This enables further decoupling between modules.

We have created a theoretical example to explain the fundamentals just mentioned above. The example is an automatic switch on/off of a light bulb based on the level of natural light. According to Figure 3.2 we need three modules. Module **Behavior** that indicates when the light is on or off. Module **Thinker** that query the knowledge base. Module **Sensors & Actuators** composed of a sensor that obtains readings of the environment and affects behavior, and a light bulb which is the actuator that performs the desired behavior.

To address this problem under the Pull approach, we use a blackboard called **Gusimplewhiteboard** [11] that uses messages of control and status for communi-
cation with its nodes. In fact, this blackboard is just a library in shared memory.

![Sequence diagram showing the execution of an on/off light switch under the Pull approach.](image)

Notice that the module Thinker is an executable program that has been compiled with a Prolog program wrapped in C-language (C_Prolog_wrapper). This executable program asynchronously receives light sensor values. We call them “sensor_status” because it is a message that is reporting the status of the sensor and it indicates if there is daylight or not. On the other side, state LIGHT OFF of the LLFSM sends a control message to the blackboard. This control message represents a desire action. Thus, the actuator gets the message from the blackboard causing bulb light turns off.

The same posted message by the LLFSM is retrieved by C_Prolog_wrapper. Also, it retrieves the status message posted by LIGHT SENSOR. With these two messages as inputs, C_Prolog_wrapper internally performs a query to the Prolog
program and writes the results on the blackboard. These results are written as status messages “shall switchState status” and the LLFSM reads them at its own pace. C_Prolog_wrapper is continuously posting results. When “shall switchState status” message matches the requirements of the state transition in the LLFSM, the state changes to LIGHT ON and the process begins again.

From this explanation Figure 3.3 emerges, which is a concrete representation of our proposed architecture using pull approach.

![Proposed architecture under the Pull approach.](image)

We can highlight from this example that the communication between modules uses non-blocking messages (control/status), allowing those components interested in information from other modules to pull (read) the communication channel at their own time. Also, we can see that coupling between messages destined to other modules is not necessary. Therefore the LLFSMs also do not need extra states to secure coupling, making the flow between states simpler.

We can contrast these observations if we build the example using the Push approach. To do this, we use ROS[35], the Robotic Operating System, as the
current representative of this pattern. However, the points we make, are applicable to any other middleware using Push approach.

### 3.2.1 Example under the Push approach

Figure 3.4 shows the representation of the example using ROS under a Push approach. Now, using ROS, the communications between modules (nodes) uses messages routed via a transport system with publish/subscribe semantics. Two kinds of messages come from this. Those messages colored in black are called ROS-topics. Those messages colored in blue/red are request/respond messages as part of ROS-services. In ROS-topics if a node is interested in a certain kind of data will subscribe to the appropriate topic. In ROS-services, a providing node offers a service and a client uses the service by sending the request message and awaiting the reply.

![Figure 3.4: Proposed architecture under the Push approach.](image)

Figure 3.5 shows the execution sequence diagram that follows a strict order so as to perform a request to the service "is_day_light" in C_Prolog_wrapper. First,
the LLFSM sends a ROS-topic message to the actuator turning the light bulb off. Subsequently, we need to get all the parameters needed by the callback function. In this case, we assume that “sensorValue” message is a parameter needed to call the ROS-service. Thus, the execution goes first with light sensor publishing an ROS-topic message. Then the subscriber LLFSM state gets the message and calls service “is_day_light”. After coupling, a respond is sent back to the LLFSM and transition fires to state TEST. Here the respond is evaluated in a loop, calling an ROS-service until the respond matches the requirements of the transition state. Finally, the state changes to LIGHT_ON.

Figure 3.5: Sequence diagram showing the execution of an on/off light switch under the Push approach.
From the above, we can observe that now the LLFSM node must collect a LIGHT_SENSOR node parameter to call an ROS-service. This process is not performed using Gsimplewhiteboard under Pull approach. In addition, due to the necessary coupling between ROS-nodes many more states are needed in the behavior representation. Similarly, we need many calls to ROS-services to evaluate transitions, thus, many coupling/decoupling processes between nodes are needed.

Although this is only a theoretical example, clearly we can see some advantages of using the Pull approach. However, the next chapter builds a real case study from which we can obtain concrete conclusions.
Chapter 4

CASE STUDY

4.1 Traffic light control problem description

This case is a very simple implementation of a traffic light control [20, section 1.3.1] at the intersection of two roads. One road goes from east to west (EW), and the other road goes from north to south (NS). Figure 4.1 shows the position of the traffic lights at the intersection.

![Figure 4.1: A traffic light control at the intersection of two roads.](image)

We have considered that system controls 6 lights, green, amber and red lights for EW-direction, and the same three colors of lights for NS-direction. The controller must ensure the system behavior. This behavior is a typical cycle in a
traffic light control. The cycle begins with GREEN_ON and then changes to AMBER_ON, and from AMBER_ON changes to RED_ON. Then the cycle repeats again. The only additional requirement is that the two sets of lights NS and EW cannot be both in the state GREEN_ON at the same time.

We know that traffic control is a very simple problem that can be solved in different ways. From our perspective, the simplicity of this problem allows the first approach of the proposal of this thesis. Therefore, this problem can be solved according to the architecture shown in section 3.2 Figure 3.1.

4.2 Modeling traffic light problem

Figure 4.2 shows a generic model of a traffic light behavior. Notice that in every state of the LLFSM the corresponding light is on, while the other lights are off. For instance if we are in GREEN_ON state the light green is on and amber and red lights are off.

![Figure 4.2: Generic model of a traffic light behavior.](image_url)

As well, notice the transitions between states in the generic model of Figure
4.2. These transitions are important topics in the solution, they constitute the base of LLFSMs and also constitute queries to the logic program. In other words, the relationship between reactive behaviors represented by LLFSM and a logic engine is defined mostly in the transitions. The way we treat these transitions is one of the contributions of this thesis.

To demonstrate the entire solution, is necessary to decompose all in small modules. Figure 4.3 shows the model of the traffic light control solution within a block diagram. This figure is an adaptation on the proposed architecture model under the Pull approach presented in section 3.2 Figure 3.3.

Figure 4.3: A block diagram composed of 7 modules of the traffic light implementation.
The modules depicted in Figure 4.3 are:

- Behavior represented by three LLFSMs: The first one is EW TRAFFIC LIGHT CONTROL, it represents the traffic light control behavior on the East-West side. The second one is NS TRAFFIC LIGHT CONTROL, it represents the traffic light control behavior on the North-South side. The third one is TIMER, it represents a timer behavior. These LLFSMs are executed by the CLFSM tool which is part of the MIPAls [17] suite tools.

- C_Prolog_wrapper: Four compiled programs; traffic_lights_ew_to_amber, traffic_lights_ew_to_red, traffic_lights_ns_to_amber, and traffic_lights_ns_to_red. These executables are those containing Prolog programs wrapped in language C. They are responsible for posting the results of the queries on the blackboard Gusimplewhiteboard. In fact, these executables are composed by three sources: Prolog, wrapper in C, and a reader/writer in C++.

- Blackboard Gusimplewhiteboard [11] embedded in the LLFSMs and C_Prolog_wrappers is inherent for communication among all modules. In fact, this blackboard is just a library in shared memory.

These modules explained above constitute the entire solution of the traffic light control problem. All concepts and theory that are part of this solution were explained in previous chapters. However, we can find further explanations for this particular case in this chapter.

4.3 LLFSMs implementation

One of the important parts of this solution is behavior represented by LLFSMs. The first two LLFSMs in Figure 4.3, EW TRAFFIC LIGHT CONTROL and NS
TRAFFIC LIGHT CONTROL are defined with three sequential states: GREEN_ON, AMBER_ON, and RED_ON. These three states can be observed in Figure 4.4 [12] for EW LIGHT TRAFFIC CONTROL. The same number of states can be observed in Figure 4.5 [12] for NS LIGHT TRAFFIC CONTROL. We can find how to build the EW TRAFFIC LIGHT CONTROL LLFSM using MiEditLLFSM tool [17] in Appendix C.

From the content in Figure 4.4 [12], we can observe different types of messages. For instance, in the OnEntry section of the state GREEN_ON_EW, we find three control messages and two status message: green_control_t, amber_control_t, red_control_t, warnEW_status_t, and reset_Timer_status_t. These messages constitute a fundamental part of the implementation of Pull approach using Gusimplewhiteboard. Control messages are referred to desired actions. Status messages are referred to the status of other LLFSMs or any other module that posts a message on the blackboard. We can find an implementation of these messages in the appendix B.

Figure 4.4: EW LIGHT TRAFFIC CONTROL LLFSM.

Focus on the GREEN_ON_EW state. On one hand, we are posting on Gusimplewhiteboard green_control_t, informing to anyone interested in this mes-
sage, that the LLFSM is on the GREEN_ON_EW state. On the other hand, we are reading from the blackboard warnEW\_status\_t which is a status message from another machine. This status message turns out to be the evaluation of a Prolog program that contains the knowledge base.

![Figure 4.5: NS LIGHT TRAFFIC CONTROL LLFSM.](image)

In particular, those status messages that come from a Prolog program evaluation trigger transitions between states in an LLFSM. In the EW LLFSM, these status messages are named: warnEW\_status\_t, stopEW\_status\_t, redNS\_status\_t. When an LLFSM need those status messages, it reads them from the blackboard. As simple as that.

All states follow the same pattern that occurred in the GREEN_ON_EW state. Likewise, it happens in the NS LIGHT TRAFFIC CONTROL LLFSM. That is, sequentially, when the state begins the performance, a control message is sent to the board stating that the corresponding light is on. Next, the state reads a status message from the blackboard. This status message comes from a Prolog evaluation. If the message value is consistent with the requirements of the transition, then the state changes and the cycle begins again.

There is also the LLFSM TIMER, as its name implies, this machine is a timer.
that publishes two messages to the blackboard: message TimeGT30 and message timeGT5. The sequence of this LLFSM begins with the seconds counter to zero. Next, a loop in which the values of the messages timeGT30 and timeGT5 are published until the reading of reset_timer_status message evaluates to true and resets the machine.

Figure 4.6 shows the behavior of the TIMER LLFSM.

4.4 C_Prolog_wrapper implementation

We have seen in the previous section how a Prolog program evaluates the transitions amongst states in the LLFSM. However, we have not mentioned the fact that the time required to query a Prolog program can vary from query to query. Even with the same knowledgebase. This fact is an important design factor of the proposed architecture. That is, we could include a Prolog program in the same arrangement of an LLFSM, but if a query takes too long, the transitions between states and therefore the system would wait a long time. Therefore, the foundations of reactive systems would be compromised, and making LLFSM work as a delib-
The above explains the necessity of another program. This one is the C_Prologwrapper. To explain it, we should know that this program is composed of three sources: Prolog, C, and C++. To begin with the first source, Figure 4.7 shows the content of the Prolog program. It is a very simple program that represents facts in a traffic light control system. Four lines determine the behavior of the lights. The first two lines indicate that if the current light is green and the value of timerGT30 is true, then the state must switch to amber state. The last two lines indicate that if the current light is amber and timerGT5 is true, then the state must switch to a red state.

\[
causeew(warnew, greenew, timerGT30).
causens (warnew, greenew, timerGT30).
causeew(stopew, amberew, timerGT5).
causens (stopns, ambers, timerGT5).
\]

Figure 4.7: Prolog program content of a traffic light control.

In fact, this program is wrapped in C language so it can be compatible with a C++ code. Readings and writings on G simply whiteboard are programmed in C++. Thus, the three sources in Prolog, C, and C++ are compiled into one executable that we name C_Prolog_wrapper.

Each C_Prolog_wrapper focuses on a particular outcome. For example, wrapper traffic_lights_ew_to_amber focuses on the first line of the Prolog program causeew(warnew, greenew, timerGT30). To fulfill the requirements of the query, traffic_lights_ew_to_amber reads the values of the messages on the blackboard posted by TIMER and EW LIGHT TRAFFIC CONTROL. Internally C_Prolog_wrapper performs a query and post a status message warnEW_status_t on the blackboard. This status message is continuously being posted on the
blackboard.

The above is the same pattern followed by the four C_Prolog_wrappers: traffic_lights_ew_to_amber, traffic_lights_ew_to_red, traffic_lights_ns_to_amber, traffic_lights_ns_to_red.

For further information, appendix A shows example of building the wrapper traffic_lights_ew_to_amber.

4.5 Sequence execution

Figure 4.8 depict a sequence execution diagram of the solution under the Pull approach. All modules of Figure 4.3 are represented in this execution. Be reminded that the blackboard Gusimplewhiteboard is just a library in shared memory embedded in each of the executables, so it has not been depicted in Figure 4.8.

The execution begins with the state GREEN_ON_NS in the NS TRAFFIC LIGHT CONTROL LLFSM. The first action is to post a green_control_t message on the blackboard. Similarly, the state GREEN_ON_NS post a reset Timer_status_t message on the blackboard. TIMER reads the value reset Timer_status_t, and the timer begins. Next, TIMER starts posting timeGT30 and timeGT5 values. The wrapper traffic_lights_ns_to_amber reads the values timeGT30 and green_control_t posted previously and internally perform a query to the Prolog program. Once the query is solved, the wrapper posts warNS_status_t. The state GREEN_ON_NS reads this value and if it matches with the transition expression requirements then the state changes to AMBER_ ON_NS.
Figure 4.8: Sequence execution diagram of the traffic light control problem solution under the Pull approach
Otherwise, the LLFSM state continues reading the status message `warnNS_status_t`, until the transition fires. The cycle is repeated in the state AMBER_ON_NS with the difference that this time, the wrapper is interested in the value of timerGT5.

Notice that while the actions of the states GREEN_ON_NS and AMBER_ON_NS are executed, the state RED_ON_EW in EW_TRAFFIC_LIGHT_CONTROL LLFSM waits for `red_control_t` in order to evaluate its transition expression. When the transition fires, the same sequence as in the NS TRAFFIC LIGHT CONTROL LLFSM occurs.

From this example, we can highlight some points.

- The LLFSM execution is performed sequentially. This is something we have already stressed in previous chapters.

- None of the components is aware of the existence of other components. Each one of them writes and reads at their convenience through non-blocking control/status messages. Therefore, there is no need for couplings between modules.

- Due to unnecessary coupling, the response time of the reactive system is reduced.

- Due to the time required to evaluate a query in Prolog vary from query to query, it is important to have C_Prolog_wrappers running in different threads, instead of having a Prolog program in the LLFSM.

In order to compare the solution proposed in this chapter, we build a solution to the problem of traffic control under the push approach. The next chapter covers this approach.
Chapter 5

CONTRASTING THE PULL APPROACH WITH THE PUSH APPROACH

To contrast the solution proposed in the previous chapter. We reconstruct the case of traffic lights control, but this time, under the push approach. The importance of this contrast lies in the more frequent use of the Push approach in robotic systems, that from our perspective, it is an approach that offers some disadvantages. To build the case under Push approach we use the Robot Operating System, ROS [24], which is by far the most used middleware in robotic systems.

To begin with the implementation, we must recall the model we used in Chapter 3 Figure 3.4. That model is the base of this construction. We use the same modules: module Behavior, module C_Prolog_wrappers, and ROS as a middleware. Remind that ROS uses messages for communication between its nodes. Those messages follow publish/subscribe semantics. Therefore, the reconstruction of this problem follow the recommendations of the ROS publish/subscriber semantic. We use both ROS-topics and ROS-services. However, the contrast we
make focuses on ROS-services.

Figure 5.1 shows a block diagram that represents the solution of the traffic light control problem under the Push approach. In order to equally compare the two versions of the solution, Push, and Pull. We have considered the same number of nodes in both Pull and Push examples. Four C_PROLOG_wrappers and three LLFSMs.

![Block Diagram of the Traffic Lights Control Problem](image)

Figure 5.1: Block diagram of the traffic lights control problem under the Push approach using Robot Operating System ROS.

Notice that communications amongst ROS-nodes are performed through ROS-services. These services use one-way transport for request/reply interactions defined by a pair of messages structures. One for the request and one for the reply. When a service is provided by a node, a callback function is needed. When a client wants to use the service, it sends a request message and awaits the reply. This interaction is analogous to a remote procedure call. TIMER LLFSM is the
one not using ROS-services. Instead, TIMER is using ROS-topics. These topics follow the same publish/subscribe semantics but the communications are considered many-to-many.

\begin{verbatim}
Bool greenNSisOn
Bool is_timeGreatherThanThirty
---
Bool recommendNSamber
\end{verbatim}

Figure 5.2: The content of the ROS-service file shallMoveNsAmber.srv.

To begin to explain communications via ROS-services which is the base of the Push approach, it is essential to know the data format of these services. The data format of the ROS-service that determines the transition between GREEN_ON state and AMBER_ON state follows a definition shown in Figure 5.2. The first two parameters, greenNSisOn and is_timeGreatherThanThirty, correspond to the request part. The recommendNSamber parameter corresponds to the response part. Thus, if a node wants to use this service, first need to fulfill the requirements of the request part. The client node needs to collect the greenNSisON and is_timeGreatherThanThirty in order to get the response recommendNSamber.

Once known the above, we can build our behavior with LLFSM.

### 5.1 LLFSMs with ROS-services

The construction of behavior using ROS-services complicates the relationship between the states of the LLFSMs. Now, we need to consider the fact that a request/response communication could fail. For this reason the number of states
increases. Figure 5.3 [12] and shows a transition between GREEN\_ON\_NS and AMBER\_ON\_NS states. Regard the significant increase in the number of states.

![Figure 5.3: Transition from GREEN\_ON\_NS to AMBER\_ON\_NS by a ROS-service.](image)

Now a simple transition between states made by the Pull approach becomes something much larger using the Push approach. Two states are added to handle the possible communication failures: TEST\_AMBAR and RE\_EVALUATE\_AMBAR. The same two kinds of states must be added to the other transitions of the entire LLFSM. In addition to, an extra state for setting up ROS is needed.

Something else we can emphasize from Figure 5 is that the LLFSM needs to collect values from TIMER before can make a request. If there were more parameters needed for building a request to a service, those have to be collected by the client of that service. This procedure contrast the modularity of the Pull approach showed in Chapter 4.

In addition, each call to a service sets up a coupling procedure between the publisher and the subscriber. Therefore, a decoupling procedure is set up as well and clearly adds more time in the execution sequence.
5.2 Execution sequence

Figure 5.4 shows the execution sequence diagram of the solution under the Push approach using Robot Operating System.

Figure 5.4: Execution sequence diagram of the solution under the Push approach using Robot Operating System.
The execution begins with LIGHT NS ROS SERVICE. The first state executes the actions related to setting up ROS and moves to the next state. GREEN_ON_NS starts communications with TIMER. TIMER, which was previously subscribed to an ROS-topic message, receives the resetTimer_status message. Next, TIMER resets the timer and starts publishing the message timeGT30_t. This message is received by LIGHT NS ROS SERVICE. Now with this information, GREEN_ON_NS can perform a call to shallmoveNStoAmber which is a C_Prolog_wrappper.

After coupling, shallmoveNStoAmber internally performs a query to the Prolog program. The result is published in the response.recommendNSamber message. That message triggers a transition to a loop. In this loop, TEST_AMBER evaluates the response received from C_Prolog_wrappper. If the response does not fit the requirements of the transition to AMBER_ON_NS, then the state RE_EVALUATE_AMBER collects again the timerGT30_t message and performs a query again by calling the service srv.amber. This loop ends when the recommendNSamber message becomes true, therefore the transitions to AMBER_ON_NS fires. The same sequence is performed between states AMBER_ON_NS and RED_ON_NS.

Notice that while LIGHT NS ROS SERVICE LLFSM is performing the above sequence, LIGHT EW ROS SERVICE is waiting in state RED_ON_EW an ROS-topic message. Once received a red_control_t message the state changes to GREEN_ON_EW and the cycle repeats again.
5.3 Case study contrast between Pull and Push approaches

We have seen so far, two execution sequences of the traffic light problem. The former, seen in Chapter 4 with a pull approach and the latter seen in this chapter with a Push approach. There are important differences between these two approaches that affect the solutions implemented in our case study. The contrast between the two solutions is pointed out below.

Related to the behavior, we have seen an increase in the number of states in the LLFSM. This increase is necessary when ROS is used under the push approach, but unnecessary when the blackboard Gusimplewhiteboard is used under the Pull approach. The need for new states under the push approach is due to necessary couplings for communication between nodes. These couplings are characteristic in semantic publisher / subscriber. The communication between publisher and subscriber uses messages that block all other actions of the LLFSM. In other words, while a subscriber requests a service, it does not perform any other action until receives a response from the publisher. However, If establishing communication fails, the request is repeated and continuous blocking other actions of the LLFSM.

By contrast, under the Pull approach and use of the blackboard, there is no need for couplings and therefore there is no need to increase the number of states in the LLFSMs. If a node requires any particular message, simply reads the message from the blackboard. If the node wants to post a message, just need to publish the message on the board without coupling with anyone. In fact, nodes are unaware of the existence of other nodes. They are aware only of what exists on the blackboard, that is to say, messages that interest them. This is an important advantage of the Pull approach over the Push approach because it significantly re-

47
duces the time of reading and writing messages. This reduction of time is directly reflected in the speed with which a machine makes decisions. Similarly, there is an advantage for the programmer since he does not need to add unnecessary states.

Related to the way a service is called when using the Push approach, a subscriber of a service has to collect all the parameters necessary for the callback function. In our case, the subscriber only collects a single parameter to query a Prolog program. However, if there is the case that Prolog requires much more parameters, the subscriber need to collect them all. For each parameter collected, a coupling is needed. In contrast with the use of the blackboard, no matter how many parameters are required to evaluate a prolog program, since parameters are simply read on the board.
Chapter 6

FUTURE WORK

We have shown in this thesis how reactive architecture represented by Logic-Labeled Finite-State Machines can be augmented with logic programming. This work amplifies the potential and possibilities of reactive architecture. However, it is not a sufficient validation for the architecture, but it has given some leads for future work. Some extensions have been thought and here are the possibilities.

6.1 Poker player and CLUEDO game player

First of all, we need to consolidate this architecture with a complex situation for a robot. This can be the case of a Poker player. The game of poker can be difficult to program in a sequential programming style or object-oriented programming styles with the classic languages like Java or C++. Prolog has better ways to classify cards in a poker hand and we think that such a program could be integrated into a finite state machine and provide sufficient evidence of the powerful of our architecture.

This kind of games are always fun and need a certain kind of intelligence to play them especially in card games where randomness plays an important role.
The unexpected situations demand a higher level of logical inference and higher numbers of possibilities. Besides, decisions have to be taken based on probabilities in almost every play of the game. LLFSM is sufficiently mature for this kind of games; we only need a very good implementation in Prolog to extend the potency of it.

6.2 Ontologies knowledge base with Prolog-LLFSM

The recent fascination for the web 3.0 and the ontologies as a reference of this web has open research works in different areas including robotics. Tenorth [29] and Palmia [23] are recent examples of integration and use of ontologies in robotics. They have created a knowledgebase based on ontologies and available online for a robot to query any other knowledge not included on the board of the robot. This case inspires our line of research. That is, use ontologies to represent a knowledgebase and query it with a prolog program and uses the responses in a reactive architecture with LLFSM.

In fact, we could combine the two options above and represent rules and actions of a by ontologies. A Prolog program would query the ontologies and send the responses to the agent.
Chapter 7

CONCLUSIONS

Throughout this thesis, we have demonstrated how a reactive system can make reasoned decisions. We integrate reactive behaviors represented by LLFSMs with declarative logic through Prolog programs. A very important thing to note is that this integration preserves the specific characteristics of the reactive agents and therefore the LLFSMs. At the same time, this integration preserves the characteristics of a logical thinking program based on Prolog.

The result of this merger is reflected in the proposal of an architecture that uses the pull approach to communicating different modules. This architecture has been tested and is functional. From my point of view, it offers advantages over other architectures, especially those using the Push approach.

This architecture is flexible and modular. You can expand the knowledge base and expand the desired behaviors. Also, it allows expanding the number of LLFSMs and Prolog programs, as many as required by the solution. The use of a blackboard permits each LLFSM or Prolog program to be updated, modified or deleted without these actions causing changes in other parts. What really matters are the messages on the blackboard.

Prolog evaluations that do not have determined times, do not cause blockages
or delays in normal operations of the LLFSMs. It is enough to put C_Prolog_wrappers in different threads and build communications using the Pull approach with control and status messages.

We have noted in Chapters 4 and 5 of this thesis points contrasting approaches Pull and Push. So I can conclude that, in the case of reactive systems, the option of using the Pull approach is the most appropriate. This is because Pull approach enables greater flexibility, has better response time, enables modularity, and does not require coupling and decoupling. Readers and writers can do their work when they decide it without having to inform others.

The extension of the LLFSM features with a Pull approach and a logic engine, have the potential to lead to significant developments in reactive architecture systems. Its fast response makes it a reasonable solution for robotic systems. As well as integrates reasoning in such a way that does not interrupt the actions of the automata but increases its power of making decisions.
Appendices
Appendix A

C_Prolog_Wrapper

A.1 Creating a C_Prolog_wrapper

The different tools used in the solution: LLFSM, CLFSM, Gusimplewhiteboard and Prolog points that in one way we have to make a query to a Prolog program from a C++ program. To this end, the solution considers to wrap a Prolog program into language C. The result is a wrapper called C_Prolog_wrapper. This wrapper uses GNU Prolog, as it offers a direct interface to language C and C++.

Looking at the transitions from GREEN_ON to AMBER_ON in the LLFSM EW_traffic_light we know that shallGOAmber is a query to the Prolog program. But, how do we do that? First of all, as mentioned above, we need to create a wrapper. To this end, let’s consider we are going to create a wrapper that contains a Prolog program, Figure 4.7. The first line is:

causeew (warnew, greenew, timerGT30).

causeew (warnew, greenew, timerGT30). In a Prolog environment, a query to the program is as simple as this:

>>causeew(X,greenew, timerGT30).

The response of this query will be warnew. This is because we are giving to the
atom causeew the true values of greenew and timerGT30. In another example, if we give to the atom false values like:

```prolog
>> causeew(X, redew, timerGT20).
```

the result will be false and nothing else.

To do this in language C. First, it is important to include `<gprolog.h>` library in the include section because we are using GNU Prolog.

The second step is to create a function in the wrapper C file (traffic_lights.c.c):

```c
_BOOL WarnEW__Wrapper(int argc, char *argv[], _Bool greenEWon, _Bool is_timeGT30).
```

Notice that we are passing two Booleans in the arguments, greenEWon and is_timeGT30. These two variables are located in the Gusimplewhiteboard. The first one concerns to the state of the East-West LLFSM, we can read it as “is the current state in EastWest green on?”. The second Boolean variable concerns to the Timer LLFSM, can be read it as “is time greater than 30?”. The function contains all the considerations presented on the website of GNU Prolog.

First we need to start Prolog with the arguments, and then using the function Find_Atom we are going to find the atom that interest us to evaluate. In this case is causeew which is the east-west traffic light fact in the knowledgebase of the Prolog program.

```c
Start_Prolog(argc, argv);
func = Find_Atom("causeew");
```

From the arguments we are going to get the Boolean values greenEWon and is_timeGT30 and pass it to the query. To do this we call the function:

```c
Pl_Query_Begin(PL_TRUE);
```

And assign to the PIterm arg[]:
arg[0] = Mk_Variable();
arg[1] = Mk_String(str1);
arg[2] = Mk_String(str2);

The assignments above can be read as argument zero is the first argument of the causeew atom and it is a variable. The arg[1] and arg[2] are strings. In this case, these strings are defined by the arguments of the function WarnEW_wrapper. Referring to the Prolog environment this is the same as to query:

```plaintext
>>causeew (X, greenew, timerGT30).
```

If we wanted to query a different argument in the atom causeew we only need to assign the MK_Variable() to the argument we want to be a variable.

After we begin the query we need actually to call the query with the function and the number of arguments like this:

```plaintext
Pl_Query_Call(func, 3, arg);
```

where “func” is the atom causeew and the number 3 is because we have 3 arguments in the atom, and then “arg” that contains the values of the arguments.

We used Pl_Query_Next_Solution() to find other possible solutions of the query and then Pl_Query_End(PL_RECOVER) to finish the query. This function is a clear picture of integration of all parts in the solution with the Gusimplewhiteboard. In other words, the wrapper file including the function WarnEW_Wrapper reads the LLFSMs outputs located in the Gusimplewhiteboard and later make a query to the Prolog program. After that post the answer into the Gusimplewhiteboard to be read by others. The answer is a Boolean value, TRUE or False.

To reproduce the entire case, we need to create other wrapper functions in C. For every wrapper, we are going to need a function. That is to say, in this case, we are going to need other three functions and follow the same procedure shown above, Figure A.1 show them.
Figure A.1: Wrapper functions.

A.2 Create a Gusimplewhiteboard writer/reader program in C++

Now it is time to build a program that reads from Gusimplewhiteboard then query to the Prolog program, and finally posts the responses of the Prolog program to the Gusimplewhiteboard.

The example we are following is the transition from Green to Amber in the East-west side of the traffic light control. To this end, we named a program in C++11 “shallMoveEWtoAmber.cpp”. In here we are going to combine everything done in previous steps. That is, we are going to use in here, the C_Prolog_wrapper named “traffic_lights_c.c” properly integrated among the Gusimplewhiteboard messages wrappered up in the file “warn_ew_c.h” and included in the INLCUDE folder of the “traffic_light_wb_Prolog” packet.

The file “shallMoveEWtoAmber.cpp” can be found in the attached CD. Here a resume of the code. First of all we need to include the messages created in the Gusimplewhiteboard, to this end, we need to include:

```c
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardtypelist_generated.h"

Also include the wrapper in C.

extern "C" {#include "traffic\_lights\_wb\_Prolog/warn\_ew\_c.h"}
```

And use

```c
using namespace std;
```
One we have done this, it is pretty simple. We are going to create a function that contains first Instances of the new classes created before. Some of them are inputs from the Gusimplewhiteboard (status of the LLFSMs states) and some of them are outputs to the Gusimplewhiteboard (responses from queries to Prolog program).

First, we create instances of the recently defined class messages to get the inputs from the Gusimplewhiteboard. This is:

```c
GreenEWOn_t green_status_t;
TimeGTthirty_t timeGT30_t;
```

Second, we create instances of defined class messages to post the responses of Prolog queries into the Gusimplewhiteboard. This is:

```c
WarnEW_t warnEW_status_t;
```

Then a simple loop performing the query to the Prolog program. First we need to get the status of the Green on state from LLFSMs East-west.

```c
greenEWOn = green_status_t;
is\_timeGT30 = timeGT30\_t;
```

Then we need to call the Prolog evaluation with the C_Prolog_wrapper function, WarnEW_Wrapper, passing the argument obtained in the previous step.

```c
warnEW=WarnEW\_Wrapper(argc,
argv,greenEWOn,is\_timeGT30);
```

After that, we need to post the returned value by Prolog into the Gusimplewhiteboard

```c
warnEW\_status_t=warnEW;
```

This is all for this program, the same principles are coded in the other C_Prolog_wrappers.
A.3 compilation using ROS Catkin

I have mentioned that this thesis is using ROS as we want to contrast the Pull approach of the Gusimplewhiteboard and the Push approach of the ROS [24]. For this reason, we use Catkin as a compiler. The documentation of Catkin tool is presented in [28].

With Catkin we build an executable program with three different sources. These three sources are those programs created above: Prolog, the wrapper in C, and the reader/writer C++. 
Appendix B

GUSIMPLEWHITEBOARD
CONTROL/STATUS MESSAGES

B.1 Definition of control/status messages in Gusimplewhiteboard

The implementation of the traffic light control considers that the communication amongst LLFSMs should be in a shared memory following a Pull approach and use control/status messages. These important characteristics are combined in a blackboard architecture in a particular shared memory object-oriented implementation of Mipal[17]: Gusimplewhiteboard [11]. In this appendix, we are going to follow the tutorial [7].

To begin with the implementation of the control/status messages in Gusimplewhiteboard first is necessary to say that although the implementation can be compiled with different compilers, we are following CATKIN [28] the recommended compiler for ROS as we are contrasting the results with this last middleware. For this reason, the first part is to create a catkin package following the CATKIN
tutorial [28]. The name of the package is “traffic_lights_wb_prolog”. Once this step is done, we proceed with the MIPAL approach to creating class-oriented C++ messages.

The first thing is to create a plain C data structure for consistent cross-platform alignment in memory. In our case is a simple Boolean so we do not have to define it, just pass to the next step. Once we have the language C structure we use it for C++11-class. One important aspect of the C++11-message is that in order to be properly integrated among the Gusimplewhiteboard messages they must be wrapped-up in a #ifndef as following: 

```c
#ifndef W_EW_WRAPPER_H_
#define W_EW_WRAPPER_H_

_Bool WarnEW__Wrapper(int argc, char *argv[],
_Bool greenEWon, _Bool is_timeGT30);
#endif
```

The name of the file for the case of East-West GreenToAmber is “warn_ew_c.h”. Again must be included in the include directory of our CATKIN package.

Now that we have a class-oriented message, it must be incorporated into the known message of the Gusimplewhiteboard. Here is where the file “guwhiteboardtypelist.tsl” and the package “guwhiteboardtypegenerator” play role. Both can be downloaded from MiPal download [17].

For our case we need to add the corresponding messages in the “gusimplewhiteboardtypelist.tsl” file as the figure B.1 shows:

Files like WBFunctor_types_generated.h, guwhiteboardtypelist_c_typestrings_generated.c, guwhiteboardgetter.cpp, guwhiteboardtypelist_generated.h, guwhiteboardposter.cpp, guwhiteboardtypelist_tcp_generated.h, and guwhiteboardtypelist_c_generated.h, are going to be replaced.

Once these lines are added, we need to execute the program “guwhiteboardtypegenerator” that is located in the compiled package guwhiteboardgenerator in
Now, however, you need to build a new Gusimplewhiteboard and its library. The easiest way is to recompile all packages. So remove everything that is being constructed and compile everything again with catkin_make.
Appendix C

LLFSM IMPLEMENTATION EXAMPLE

C.1 Building behavior with MeditLLFSM editor

We now construct the first part of the traffic light control LLFSM. Once downloaded the MeditLLFSM [17] we simply start it. The example we are going to follow is the state of the East-west traffic light RED_ON_EW. The edition screenshot is shown in Figure C.1

We can easily see that the interface is very intuitive. In the tab VARIABLES, we declare all useful variables. In the tab INCLUDES, we Include all we need. In the OnEntry section we put the code we want to execute when entering to this state. In the OnExit section we put the code we want to execute when leaving the state. In the Internal section we put the code we want to execute only if none of the transitions fires.

It is necessary to show the content of the different tabs to understand the construction of this state in the traffic light LLFSM. Figure C.1 shows the tab of the States. From here we can move between states in order to modify the state.
In this case, as explained before, we have 3 states and one pseudo-state. The 3 states are RED_ON_EW, GREEN_ON_EW, AMBER_ON_EW. Figure C.3 shows the variables used in the entire LLFSM. Here is important to notice that we are using non-general types of variables but using also generic types defined in the GUsimplewhiteboard in an object oriented implementation.

We can see in here 3 Boolean variables, shallGoRedEW, shallGoGreenEW, shallWarnEW. These 3 variables are used in the transitions from states to states. In Figure C.4 we can see the use of the variable shallGOGreenEW, that is when the current state is RED_ON_EW the transition will fire to the target GREEN_ON_EW.
Following the OnEntry section in Figure C.1, we can see that the LLFSM gets the status message red\_NS\_status\_T from the whiteboard and assign the value to the variable shallGoGreenEW which is the condition of the transition expression. Opposed to the get message, the LLFSM are posting its values through the control messages, e.g. green\_control\_t is posting a FALSE value. Amber\_control\_t is posting a FALSE value. red\_control\_t is posting a TRUE value. That is, we are posting to the whiteboard that the only bulb RED of the traffic light East-west is ON.
Bibliography


