Verification-driven design and programming of autonomous robots

by

Paolo Izzo

A thesis submitted as partial fulfilment for the degree of Doctor of Philosophy

The University of Sheffield

Faculty of Engineering

Department of Automatic Control and Systems Engineering

26th March 2017

Abstract

This thesis describes a new agent-based architecture called the Limited Instruction Set Agent (LISA). Agent-based systems are a popular approach to the implementation of autonomous behaviour, and they usually consist of a 'reasoning' module that commands lower level subsystems that in turn interact with the environment. When an autonomous system is placed in any environment, the correctness of the software must be guaranteed for safety. This is generally done with 'verification by model checking' which consists of creating a model, which represents the system and its interaction with the environment, and then proving specifications using the model. Most agent frameworks to date do not contemplate verification as a design feature and they generally share a few drawbacks: the generation of a model that can be verified by a model checking software is either done manually or by executing the agent code recursively and exploring every possible path to list the state space of the system. The LISA system is based on existing agent-based architectures and it is designed to be structurally simpler than its predecessors with the aim of facilitating the verification process. The agent program of LISA is enriched with structures that allow to model the probabilistic nature of environmental events, so that they can be taken into account in the verification process. The LISA program can be automatically translated to a verifiable probabilistic model suitable for verification with existing software tools such as PRISM. Furthermore, the system is structured to minimise the size of its probabilistic model, and ultimately offers a faster verification process. The thesis contains a number of theoretical contributions to the LISA programming system, including run-time verification for prediction of future outcomes of actions, and the new methods are illustrated on the programming and simulation with an example of autonomous surface vehicle for sea mine detection and disposal.

To my parents Antonio and Carmela,

and to my beloved wife Teresa.

If I had nothing else but their love,

I would still be the richest man.

Acknowledgments

This research project was funded by Thales UK and The University Of Sheffield. I would like to thank my supervisor Prof. Sandor Veres. I would also like to thank Dr Hongyang Qu, senior researcher at The University Of Sheffield, for the help with verification theory and with verification software, and all the academic and administrative staff of the Automatic Control & Systems Engineering for providing me with a friendly and stimulating environment. Last but not least a thank you also goes to Tim Munn and Leslie Hart from Thales UK for their support and stimulating interest.

Contents

List of Figures

List of Tables

Acronyms

AET Agent Executive Toolbox. 104

AIL Agent Infrastructure Layer. 4, 20

AJPF Agent Java PathFinder. 5, 20, 21, 82

AOP Agent Oriented Programming. 4, 16, 17, 20–22, 38, 56

API Application Programming Interface. 104, 112

ASV Autonomous Surface Vehicle. 115–117, 119, 128, 129, 133

BDD Binary Decision Diagram. 123

BDI Belief-Desire-Intention. 4–6, 12, 13, 15, 17, 20, 37, 39, 42, 47, 48, 68, 73, 74, 80, 96, 99, 101, 102, 118

BRF Belief Review Function. 51

BUF Belief Update Function. 49, 50, 54

CAT Cognitive Agent Toolbox. 17, 43, 102, 104, 106, 109, 112, 133

COG Centre Of Gravity. 126, 137, 139

CTL Computation Tree Logic. 24

CTMC Continuous Time Markov Chain. 28

DOF Degree Of Freedom. 31, 130, 133, 137

DTMC Discrete-Time Markov Chain. 6, 7, 23, 24, 26–28, 63, 71, 72, 74, 77–81, 95, 99, 122–125, 129, 131–133

FIFO First In, First Out. 44

GPS Global Positioning System. 117

HS Hybrid System. 3, 16, 33

IvP Interval Programming. 102, 103, 108–113, 133

JPF Java PathFinder. 4, 5, 20, 21

LISA Limited Instruction Set Agent. 5–8, 16, 29, 37–40, 42–45, 47–49, 51, 52, 55–58, 60–62, 64, 65, 67, 68, 71–82, 84, 89, 92, 93, 95, 96, 98, 99, 101–110, 112, 115, 118, 119, 122, 128, 129, 131–135

LTL Linear Temporal Logic. 5, 20, 21, 32–34

MDP Markov Decision Process. 6, 7, 23, 26–28, 63, 71, 72, 74, 77, 79–81, 85, 86, 95, 98, 99, 122–125, 129, 131–133

MOL Machine Ontology Language. 18, 104

MOOS Mission Oriented Operating Suite. 102, 103, 108–113, 133

MTBDD Multi-Terminal Binary Decision Diagram. 123

NLP Natural Language Programming. 6, 18, 35, 38, 43, 56, 81, 103, 117, 132

OOP Object Oriented Programming. 3, 16

PCTL Probabilistic Computation Tree Logic. 24–26, 28, 33, 34, 72, 73, 81, 93, 94

PI Proportional-Integral. 128

PRS Procedural Reasoning System. 4, 13, 17

PTA Probabilistic Timed Automata. 24, 28

ROS Robot Operating System. 102, 103, 108, 109, 112, 133

RRT Rapidly-exploring Random Tree. 32, 33

sEnglish system-English. 6, 7, 18, 38, 43, 46, 47, 56, 57, 59, 60, 68, 81, 82, 85, 88, 90, 93, 102–104, 106, 109, 112, 117, 120, 132

SLAM Simultaneous Localization And Mapping. 30

VR Virtual Reality. 125, 129, 133

List of Symbols

Autonomous Agents description

Λ	Set of all possible indices
λ_j	Index for plan j
$\mathcal{R}$	Rational BDI agent
Π	*Plan library*
π_j	Plan j
$\boldsymbol{\lambda}$	Set of plan indices
A	Set of *actions*
B	Set of *Beliefs*
B_0	Set of *initial Beliefs*
B_a	Set of *action feedbacks*
B_m	Set of *mental notes*
B_s	Set of *sensory beliefs*
D	Set of *Applicable Plans*, or *Desires*
E	Set of *Events*
e_j	Event j
F_O	*Plan* or *Option* selection function
f_P	*Plan unification* function
F_{act}	*Action execution* function
f_{BR}	*Belief Review* function
f_{BU}	*Belief Update* function
I	Set of *Intentions*

n_π	Number of plans
n_e	Number of events
p	Percept

Formal Verification

$\mathcal{D}$	Discrete Time Markov Chain (DTMC)
$\mathcal{M}$	Markov Decision Process
$\mathtt{F}$	Eventually
$\mathtt{P}_{\bowtie p}$	Probability operator
$\mathtt{R}$	Reward operator
$\mathtt{U}$	Until
$\mathtt{X}$	Next

Chapter 1.

Introduction

IN recent years there has been an exponentially increasing interest in the field of autonomous robotics especially driven by the increasing quality and depth of autonomy that current hardware and software technology is being capable to produce. From self-maintenance and indoor navigation with consumer domestic robots such as iRobot products [39], to construction [7], to advanced task performances for scientific [38] and military [97] applications, researchers and developers have been able to produce robots with a spectrum of degrees of autonomy, giving autonomous robots an important role in modern society.

Given the large community working on the topic, it should not be surprising that an incredibly large variety of hardware and software architectures exist and are constantly developed. A very popular approach to the implementation of software for autonomous robots is that of creating modular architectures with different layers of abstraction, from modules that operate at lower level, closer to the physical world, to modules that generate abstract plans of action to direct the lower level modules. A software architecture that reproduces some sort of autonomous behaviour is usually referred to as an *autonomous agent*.

Although there exist systems that can be designed to guarantee the correctness of specific behaviours under all circumstances, generally speaking, for a system to be considered safe enough to be placed in a real-world scenario, verification and validation of some sort is always needed. Software in general, and autonomous agents in particular, are no exception to the rule.

This thesis focuses on the design, implementation and simulation of autonomous agents that are intrinsically predisposed to be verified with popular and well acknowledged verification tools. The idea is to set a framework where the program that describes the decision-making of the autonomous agent is developed in a easy to grasp language, that at the same time allows to include enough information to automatically generate an abstract, complete model of the system, that can be verified by dedicated, widely recognised software. Furthermore this thesis explores the use of verification techniques to give an additional degree of knowledge to the agent and therefore improve the performances of the decision-making engine itself.

1.1. Related work and motivations

Autonomous control is a branch of control science that emerged as an evolution of classical *feedback control* [11, 70]. The purpose of feedback control is to regulate a system in order to make it follow a reference input. Traditionally the controller relies on an external reference, and the controller itself does not have any decisional power over the reference. This is where *autonomous* controllers come in: they are designed to make decisions on *what* control reference to use and, more generally, what goals to achieve and *how* to achieve them. They do so by sequencing plans from a pool of available actions, considering their current understanding of the state of the world [10]. This gives autonomous controllers a high level adaptivity, an ability to act appropriately in a variety of environmental situations, under a variety of level of uncertainty, a property that is sometimes referred to as “intelligence” [161].

Given the ever increasing affordability and computational power of hardware and the level of connectivity offered by the internet, which makes the formation of large communities working on the problem more possible than ever before, most innovation in autonomous decision-making is likely to come from dedicated software. A first attempt towards software for autonomous decision-making was initially made by using Object Oriented Programming (OOP). Early examples of development in this direction can be found in [166, 227]. More recently in [186], Ridao presents a layered OOP control architecture with deliberative, control execution and reactive layers. For autonomous control, OOP frameworks are mostly associated with Hybrid Systems (HSs) modelling, where the term *hybrid* refers to the use of continuous time dynamics switched by discrete state transitions in a unified framework [6, 207]. A few examples of this trend can be found in [14, 171, 195]. Hybrid systems are a very broad and highly general class of system models to be directly applicable in problems of decision-making, though most robotic autonomous systems can ultimately be represented as hybrid systems under uncertainty.

Objects in OOP are generally passive, in the sense that they only operate when their methods are called by an external function. This behaviour is a definite limiting factor for the development of truly autonomous decision-making software, which led to the development of new decision-making architectures called “autonomous agents” [213, 224], which feature software that share some similarities with objects but work with a substantially different approach [224]. Agents have a significantly greater degree of control over their own internal state and have *active* components, in the sense that they actively execute actions in order to move closer to a goal. A formal description of autonomous agents can be found in [213, 224, 225].

An autonomous agent is commonly described as a two part system [189]: the *agent architecture* and the *agent program*. The agent program is a function that maps sensory information to actions, and the architecture is a description of how this function interfaces with lower level subsystems, also known as *skills*, and how these lower level subsystems communicate with each other. Most robotic agent architectures are structured in a layered way, as described in [2, 88, 196]. Different levels of interaction can be defined between the layers, which can vary based on the spectrum of functionality that the developer intends to implement.

One of the most widely used “anthropomorphic” approaches - that is the implementation of behaviour that mimics the way humans make decisions - to the development of autonomous decision-making is the Belief-Desire-Intention (BDI) architecture [37, 213]. Two of the most widely known implementations of the BDI architecture are the Procedural Reasoning System (PRS) [91, 92] and *AgentSpeak* [181]. The latter fully embraces the philosophy of Agent Oriented Programming (AOP) [193], and it offers a Java based interpreter that can be customised according to the designer’s needs.

As for any system that aims to be introduced in real-world situations, autonomous controllers need to be verified against publicly acceptable standards. *Formal verification* is a great tool to do so and it is traditionally carried out in one of two ways: axiomatic, verifying mathematical models of system by theorem proofing, or semantic, verifying numerical models of the system [224]. The most accessible and widely used approach to the verification of autonomous system uses a semantic approach, with *model checking* [52]. In particular *probabilistic* model checking [134] is used to analyse probabilistic models by checking wether or not a given specification holds true in the model.

The scientific community has produced many attempts to the verification of agent-based software architectures over the years. Some early examples can be found in [22, 183], and more recently with [33–35].

A subsequent effort towards verifiable agents was made by Dennis *et al.* [60] with a BDI agent programming language called *Gwendolen*, which is implemented in the Agent Infrastructure Layer (AIL) [61, 62], a collection of Java classes intended for use in model checking agent programs, particularly with Java PathFinder (JPF).

An evolution of JPF is Agent Java PathFinder (AJPF) [64], which is built on top of JPF but specifically designed to verify agent programs, also using a Linear Temporal Logic (LTL) [177] based specification language. However JPF and AJPF introduce a significant bottleneck in the workflow as the internal generation of the program model, which is created by executing all possible paths, is highly computationally expensive. In [117] it is proposed to alleviate this problem by using JPF to generate models of agent programs that can be executed by other model-checkers. This idea is further developed in [63], which shows how AJPF can be modified to generate models in the input languages of Spin [113] or PRISM [135]. The latter is a *probabilistic* model checker, which is very important when applied to real-world applications: the probabilistic nature of events and sensed measurements requires the adoption of probabilistic modelling and verification. The work presented in [63] does describe a technique to model the full system with a probabilistic model, however a computational cost problem still remains as AJPF explores the entire execution space of a symbolic model of the agent code. Furthermore the programmer is required to implement the probabilistic model by modifying the methods of a specialised JAVA class, making the process less accessible to users that are not familiar with JAVA programming.

1.2. Contributions

The contribution of the work presented in this thesis is the investigation of a novel agent architecture called Limited Instruction Set Agent (LISA) [118]. This new architecture is based on the BDI paradigm, and it is structured as a three-layer architecture [88], with agent reasoning on top, a sequencing middle layer and a sensing and feedback control layer, with symbolic communication between all layers. The agent reasoning is based on previous implementations of AgentSpeak such as Jason [36, 37], and it is designed to facilitate development of complex agents while allowing for automatic verification. All modifications made from Jason, for the new agent architecture, have been made for one or both of the following reasons:

- *Simplify the syntax of the agent program* while maintaining the required level of expressibility for the development of rational agents.
- *Reducing the size of the state space* of the model required to abstract the agent while maintaining the required level of decision-making power offered by other BDI agent implementations.

These modifications ultimately lead to an agent reasoning that is more understandable for users and it is easier to verify. This thesis proposes a method to automatically generate from the agent code a probabilistic model for verification with the probabilistic model checker PRISM [135, 205]. In particular, the LISA reasoning is proven to be modellable as a Discrete-Time Markov Chain (DTMC) or Markov Decision Process (MDP) depending on the particular application.

The agent program is developed and described with SYSTEM-ENGLISH (sENGLISH) [146, 211], in a Natural Language Programming (NLP) interface that ensures conceptual clarity of agent decisions, sharing of programming knowledge in a team of developers and also to define shared understanding between a human operator and an autonomous system, in terms of world model items, their relationships and related actions. In this work, a few additions are made to the sENGLISH agent program to enable probabilistic modelling of environmental variables. With these modifications the user will be able to include in a single, unified document, the agent program and all the necessary information to automatically generate a probabilistic model of agent reasoning for verification. In particular probabilistic models are proposed for different kinds of environmental variables so to allow the user to include within agent logic, a finite set of paramenters that define probability distribution to describe the interaction of the agent with the external world. Although this approach still requires the user to define the probability distributions, it represents an innovative tool to facilitate and encourage formal verification of autonomous agents.

The automatically generated probabilistic model is also shown to be useful to improve the nondeterministic decision-making capabilities of the agent with a run-time verification process. This gives the agent the ability to use a probabilistic model and model checking tools to look into the consequences of choices, and deliberate on the probability of success/failure of applicable plans before committing itself to execute one.

1.3. Structure of the thesis

The thesis is organised as follows. Chapter 2 gives an overview on what is the definition of agent in this context, in Section 2.1, it introduces the topic of verification of autonomous agents in Section 2.2 and it also gives an overview of the skills and algorithms that can be used with the agent reasoning to achieve autonomy in robotic systems in Section 2.3. Chapter 3 describes the architecture of the new agent-based system LISA in Section 3.2, the agent reasoning in Section 3.3, described by highlighting differences and new features compared to Jason, and the agent program in Section 3.4 including the new features that allow the developer to include a probabilistic model of environmental variables. Chapter 4 reports the process of verification of the LISA reasoning by first proofing its modelability as a DTMC or MDP in Section 4.2 and then describing the process of converting the agent program expressed in sEnglish to a model described in the input language of Prism in Section 4.3. The process of using the verification as a design-time tool is described in Section 4.4 and as a mean of predicting future outcome of actions for better plan selection at run-time in Section 4.5. Chapter 5 gives an overview of how the LISA system can be implemented using existing tools in the robotics community and how it can be integrated with existing algorithms. Finally Chapter 6 presents a case study with a possible implementation and simulation of a LISA system.

Chapter 2.

Background

This chapter gives an overview of the main topics touched upon throughout this thesis, in order to introduce the reader to related basic definitions and concepts. Autonomous agents are first described as a general concept with definitions and principles of implementation, followed by basic concepts on verification, and in particular verification of autonomous agents. An overview of possible skills and algorithms to be later implemented within the agent framework is also presented.

2.1. Autonomous agents

A system that operates independently from human intervention, can be mainly classified as *automated* or *autonomous*. A closed loop, automated system, traditionally referred to as *feedback system* [11, 70], features a controller that regulates the input to a dynamical system so to change the output of the latter and match it with a reference signal. However the control system itself does not have any power over the reference input. A common (probably overused) example of this is the steam engine: a mechanical device called *fly-ball governor* mounted on the shaft, uses proportional control to regulate the heat, and therefore the rotational speed of the shaft itself; however the reference rotational speed is regulated by a human operator. *Autonomous* systems [10, 161, 220, 223] on the other hand, have a certain level of self-government over their own internal state and reference signal; in other words they do not require an external entity to set the reference signal, but they set it autonomously according to the design objectives. Autonomous systems are designed to perform under significant uncertainties for extended periods of time, without external intervention [5]. This is certainly a very broad definition, that covers a full spectrum of systems from *low degrees* of autonomy, where they can tolerate a restricted range of disturbances, to *higher degrees* of autonomy, where the system plans its own action and performs them unless revoked by an operator, to *full* autonomy, where the system is completely independent from human control. A review of levels of autonomy in unmanned vehicles can be found in [212].

Generally speaking any system that shows some level of autonomy could be considered an *agent*. Unfortunately there is no general consensus beyond the fact that the definition of *agent* is strictly correlated to that of *autonomy*. A general definition, given in [224, 225], is adapted here as follows.

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous actions* that can *influence* the environment in order to meet its design objectives.

In the definition above, *environment* is everything that is external to the agent, sometimes also referred to as the *world*. In [189], Russel suggests an interesting classification for types of environments, which are described to be (or not be) *accessible*, when the agent is able to sense the complete state of the environment, *deterministic*, when the next state can be determined by looking at the current one, *discrete*, when the state space is countable.

In case of mobile robots in real-world scenarios, the environment will usually be *inaccessible*, *nondeterministic* and in most cases *discrete*, assuming that discretisation can be performed in a meaningful way for the agent. Inaccessibility is due to the fact that the sensing equipment of the agent is inevitably limited compared to the amount of information available in a real-world scenario. For this reason the environment is therefore nondeterministic to the "eyes" of the machine, which implies that when the agent performs the same action twice, the latter will not necessarily produce the same outcome, it will not bring the environment and/or the agent to the same state.

This level of uncertainty is most common in real-world scenarios, and it is a significantly hard problem to approach with a classical feedback controller. The state of the world hardly reducible to a single or even a small number of variable to be regulated to match a reference signal, which makes the *proactiveness* of autonomous agents a desirable trait.

In [225], Wooldridge *et al.* argue that an autonomous agent shows the following properties.

Reactiveness The agent is able to perceive changes in the environment and it acts accordingly in a timely fashion.

Proactiveness The agent displays *goal-directed* behaviour: it does not wait for a change in the environment to happen, but it *takes the initiative* in order to meet its goals.

Autonomy The agent is able to operate without human intervention, and it owns a certain degree of control over its internal state.

Social Ability The agent is able to communicate with other agents and possibly with humans.

An agent-based system can be described by defining two main characteristics: the *architecture* and the *agent program* [189]. The agent program is a function that implements the agent mapping from percepts to actions. The architecture is the structure that describes how the agent program interfaces with lower level control subsystems and ultimately with the environment. In [153] the architecture is described as 'the backbone of robotic systems': different architectures reflect the agent program in different ways, so choosing the right architecture for the particular application is a crucial step in agent-based systems development.

Over the years the research community has produced many agent architectures (see references [210, 213] for a systematic overview), including: purely *logic-based* [1, 81, 143], *behaviour-based* or *situated* [8, 42, 152, 187, 188], *situation calculus* [57, 79, 144], *Belief-Desire-Intention (BDI)* [93, 182, 184]. These architectures are not entirely distinct, as definitions often overlap. Most modern architectures are structured in a *layered* way, where layers essentially represent abstraction levels, as described in [88, 163], with some practical examples in [2, 164, 196, 216].

Figure 2.1 shows the general structure of any agent-based system architecture. The *agent reasoning* or *agent logic*, is connected to other control systems with generally lower levels of abstraction, which are referred to as *skills.* The way the agent *acts* on the environment is by issuing action commands to its skills which can in turn apply desired changes to the output. In the same way, the agent gathers information about the world through *perception* skills, which convert numerical information coming from physical sensors into something that an agent program can work with, namely symbolic Boolean variables or discrete structures.

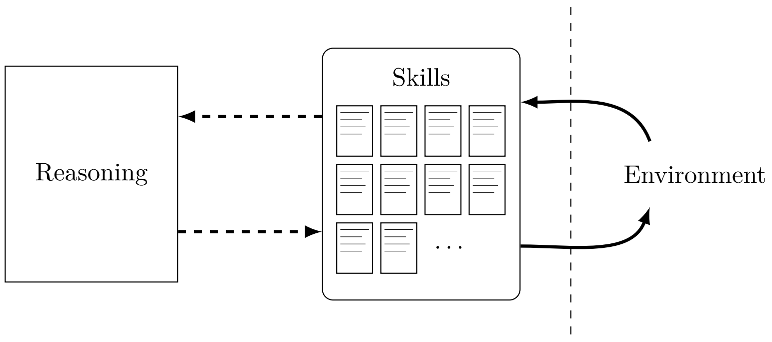


Figure 2.1.: *Generalised structure for an agent-based control system: the agent reasoning interfaces with the world through specialised functions called skills.*

The agent system described in this thesis is mainly based on the Belief-Desire-Intention (BDI) architecture, which is possibly one of the best known and studied model of reasoning agents. As the name suggests BDI agents are characterised by three large sets of symbolic information: *Beliefs*, *Desires* and *Intentions*. The Beliefs set represents the information the agent has about the world, the Desires set contains optional actions that the agent *might* want to accomplish and the Intentions set represents the set of options that the agent is committed to work towards. The model was first proposed by philosopher Michael Bratman [40, 41]. Early implementations can be found in [90–92] with the PRS, which has been re-proposed and re-implemented several times during the 90's [44, 67, 115]. The approach taken here is slightly different from that of PRS as plans are assumed to be linear sequences of actions rather than the hierarchically structured collections of goals of PRS. An interesting discussion on the role of plans in practical reasoning can be found in [178, 179]. A formal definition of BDI agent and a brief explanation of how it operates is given in the next Subection 2.1.1.

2.1.1. Formal definition

Here follows a formal definition of a generic *rational BDI agent* [213, 224]. The term "rational" indicates that the agent carries out some logic based reasoning as part of its normal functioning.

Definition 2.1 (**Rational BDI agent** [118]). *A* rational BDI agent *is a tuple*

$$\mathcal{R} = \{\mathcal{F}, B, B_0, L, A, A_0, \Pi\}$$

where:

- $\mathcal{F} = \{p_1, p_2, \ldots, p_{n_p}\}$ *is the set of all atomic prepositions that can represent beliefs or actions.*

- $B \subset \mathcal{F}$ *is the* Beliefs set*, the set of all beliefs available to the agent. A* Belief *is an atomic prepositions that represents an abstract concept.*

- B_0 *is the* Initial Beliefs *set, the information about the world that is available to the agent at the first iteration.*

- $L = \{l_1, l_2, \dots l_{n_l}\}$ *is a set of logic-based implication rules on the predicates of* B*. These rules help the agent give additional meaning to the set of current beliefs.*

- $A = \{a_1, a_2, \dots, a_{n_a}\} \subset \mathcal{F} \setminus B$ *is a set of all available actions. An* action *is an atomic preposition that is associated to a function, which acts either on the environment or on the internal state of the agent.*

- $A_0 \in A$ *is the set of initial actions.*

- $\Pi = \{\pi_1, \pi_2, \dots, \pi_{n_\pi}\}$ *is the set of executable plans or* plan library*. Each plan* π_j *is a sequence* $\pi_j(\lambda_j)$*, with* $\lambda_j \in [0, n_{\lambda_j}]$ *being the* plan index*, where* $\pi(0)$ *is a logic statement called* triggering condition*, and* $\pi_j(\lambda_j)$ *with* $\lambda_j > 0$ *is an action from* A*.* □

Each triggering condition for the plans in Π is composed by two parts: a *triggering event* 'e' and a *context* 'c', and it is usually express in the form '$e : c$'. An *event* is a belief paired with either a '+' or a '$-$' operator to indicate that the belief is either added or removed. By defining the plan library, a set

$$E \subseteq B \times \{+, -\} \tag{2.1}$$

of events is implicitly defined by the set of all triggering events. The *context* is a logic condition that the agent verifies against the current Beliefs when a plan is triggered. The expression

$$B \vDash c \tag{2.2}$$

signifies that the Beliefs set B "satisfies" a logic expression 'c' on predicates from $\mathcal{F}$, or in other words when the conditions expressed by 'c' are true on B.

Actions from A can be either *internal*, when they modify the Current Beliefs set to generate internal events, or *external*, when they are linked to external functions. Beliefs generated by internal actions are also called 'mental notes'.

Usually the agent program of a BDI agent is operated through indefinitely repeated cycles called *reasoning cycles*. To facilitate the description of the reasoning cycle of the agent, the following definition introduces dynamic subsets of the sets of Definition 2.1 that are regularly updated throughout the agent operation.

Definition 2.2 (**Operational sets of a rational BDI agent**). *Given a rational BDI agent $\mathcal{R}$, if 'time' $t \in \mathbb{N}_{\geq 1}$ is the integer count of reasoning cycles:*

- $B[t] \subset B$ *is the* Current Beliefs *set, the set of beliefs available at time t. Beliefs in $B[t]$ can be negated (usually with a '$\sim$' symbol).*
- $E[t] \subset E$ *is the* Current Events *set, which contains events that are active at time t.*
- $D[t] \subset \Pi$ *is the* Applicable Plans *or* Desires, *which contains all plans π_j such that $B[t] \vDash \pi_j(0)$.*
- $I[t] \subset \Pi$ *is the* Intentions *set, which contains plans π_j that the agent is committed to execute, for which $\lambda_j > 0$. Any plan stays in the Intentions set until all the actions listed in it have been executed, unless a* plan withdrawal *action is issued to cancel the plan.* □

For most BDI agent architectures the *reasoning cycle* is operated as follows. At the beginning of every cycle, $B[t]$ is updated by checking for external inputs and internal actions; from the changes that happen at each reasoning cycle to the Current Beliefs, a set of events is generated and added to $E[t]$. The plan library is then searched for plans that feature a triggering condition that satisfies the Current Beliefs set ($B[t] \vDash \pi(0)$). These plans are then copied to $D[t]$. A single plan from the Desires set is then selected for execution and pushed into $I[t]$. Then the agent takes applicable actions from the active plans in $I[t]$ and execute one (or more) of them. At this point the cycle is complete and $B[t+1]$ is generated. A detailed mathematical description of the LISA reasoning cycle is given in Section 3.3.

2.1.2. Agent oriented programming

As the name suggests, Agent Oriented Programming (AOP) is a paradigm for describing and implementing agent programs. AOP was initially developed as an evolution of Object Oriented Programming (OOP) [219] as described in [193, 225]. Early examples of autonomous decision-making software that uses OOP can be found in [85, 166, 226, 227]. More agent-related approaches can be found in [18, 19], with the JAVA-based framework JADE, and [186], which presents a OOP-based layered architecture. For autonomous control, OOP frameworks are mostly associated with Hybrid Systems (HSs) modelling, where the term 'hybrid' refers to the use of continuous time dynamics switched by discrete state transitions in a unified framework [6, 207]. A few examples of this trend can be found in [14, 46, 170, 171, 195]. The definition of hybrid system covers a large variety of systems, it is not specific to problems of decision-making, in fact most robotic systems can be represented with HS models.

While there are clear similarities, there are also fundamental differences between the concept of agent and the concept of object [224]. Objects are generally passive, in the sense that their methods are activated by external calls, for instance from other objects, but they do not incapsulate the choice of action. An agent on the other hand can be *requested* to perform an action, but it will only do it if it is in line with the set of beliefs at the time of request. In other words in the object-oriented case, the decision of executing an action lies within the object that invokes it, while in the agent case it is the agent itself that makes the decision. The key idea of AOP is to implement agent programs in terms of high level symbolic information, such as beliefs, desires, and intentions.

Some popular software packages and languages for AOP include

- PRS [90–92]. Precursor to BDI architectures, it provides a declarative semantics for the representation of knowledge and an operational semantics which connects the knowledge to goals.

- AgentSpeak [181]. A BDI based architecture for development of agent systems, meant to be an abstraction of existing systems such as PRS.

- Golog [59, 79, 144]. It provides an interpreter that maintains a representation of the environment being modelled, assuming the user explicitly defines conditions of actions on the environment. 3Apl [105, 106]. It offer an operational semantics that is defined with transition systems.

- Jack [114, 167]. A multi-agent system development architecture developed in Java. It builds upon PRS and dMARS [67].

- Jason [36, 37]. An evolution of AgentSpeak, developed in Java and it allows the customisation of most aspects of the agent system.

- Goal [16, 202]. BDI based, it focuses on the interface wiht the environment.

- Pddl [94, 157]. An agent programming language mainly inspired by Strips [80] and Adl [169], in an attempt to define a common formalism for describing planning domains.

One of the most complete frameworks for AOP is the Cognitive Agent Toolbox (CAT) [214], which integrates the capabilities of multiple external software suites

(Matlab, Simulink, Mcmas) and supports the development of agent reasoning with Natural Language Programming (NLP) in a language called sEnglish [146, 211]. sEnglish uses natural language sentences in an easily readable document so that even an untrained human operator can make sense of the reasoning process of the agent. An sEnglish document is organised in a *reasoning file* and multiple *action files*. The action files are descriptions of actions that can be implemented in different programming languages or as a sequence of sEnglish sentences and that are associated with a predicate that can be used anywhere in the reasoning file. Similarly to Jason, a sEnglish reasoning file is structured in sections as follows: `INITIAL BELIEFS AND GOALS`, `INITIAL ACTIONS`, `PERCEPTION PROCESS`, used to configure objects for world modelling, `REASONING`, where the logic-based implication rules are listed, and `EXECUTABLE PLANS`, the Plan Library.

sEnglish also supports the definition of an ontology, a structure that associates atomic prepositions to common data types or structures of data types. The user can then use these atomic prepositions within the agent program in order to improve clarity and readability. The language used to define the ontology in sEnglish is called Machine Ontology Language (MOL). A simple example is shown in Figure 2.2.

```
>location
  @coordinates: vector
  @covariance: matrix
    >>waypoint
      >>>global waypoint
      >>>local waypoint
```

Figure 2.2.: *Example of type definition of ontology using Machine Ontology Language.* Classes *are indicated by a single '>' symbol,* subclasses *by multiple '>' symbols and attributes by the '@' symbol.*

Although it does not influence the verification process in this particular implementation, the definition of aliases for file types in the ontology specification is a feature that allows the user to better integrate the agent reasoning program with its skills. In the implementation of the verification process presented in this thesis, the software does not take into consideration the value of a variable itself but the belief it is associated to.

2.2. Agent verification

Autonomous agents have a considerable potential for implementation in autonomous control systems. The great flexibility of the software programming gives the designer a great deal of freedom to encapsulate a variety of decision-making capabilities. However the introduction of autonomous agents in real-world scenarios brings along safety concerns, as for example highlighted in [3], especially in applications such as spacecraft control [101, 147]. In order to guarantee safety and improve people confidence in autonomous agents, the system must be certifiable against publicly accepted standards. Verification of a system is the process of checking whether or not its implementation is correct with respect to the original specification. Approaches to verification of software systems can be divided into two broad classes: *axiomatic* (deductive) and *semantic* (model checking) [224].

Axiomatic verification consists of deriving a logical theory that represent the behaviour of the agent program and formally proving that this logical theory reflects the original specification of the program. In other words, once an abstraction of the system has been created, the verification consists of a proof solving. Axiomatic verification was pioneered in the late 1960s [108] and a few examples of application to autonomous agents can be found in [154, 222], and later in [103, 104] where the authors use structured operational semantics [176] to axiomatise their 3APL language. Verification by model checking on the other hand, is applied to a model of a system, typically a finite-state machine, to check whether or not a specific temporal logic [132] formula holds true in the model. *Probabilistic* model checking analyses probabilistic models such as Markov chains and Markov decision processes [190], with specifications that are probabilistic extensions of temporal logic. A more detailed description of the process is give in Subsections 2.2.1 and 2.2.2.

Some of the most popular model checking software include: Smv [53, 160], Spin [111–113], Uppaal [17], JPF [215] and Prism [107, 135]. Popular model checkers that are specific to agent verification are AJPF [64], an evolution of JPF that uses a LTL based specification language extended with descriptions of beliefs, intentions etc., and Mcmas [149], which specialises in verification of multi-agent systems. AJPF (and JPF) are 'program' model checkers, which means that they operate on the agent code, rather than on a model of the program's execution, usually the case for traditional model checkers.

An early attempt to verification by model checking of BDI systems can be found in [183], and similar algorithms can be found in [22]. In [32–35], the authors present an automatic translation software from the AOP language *AgentSpeak* into either Promela or Java, and then use the associated model checkers Spin and JPF respectively.

A subsequent effort towards verifiable BDI agents was made by Dennis *et al.* [60] with a BDI agent programming language called *Gwendolen*, which is implemented in the AIL [61, 62], a collection of Java classes intended for use in model checking agent programs, particularly with JPF. However JPF introduces a significant bottleneck in the workflow as the internal generation of the program model, which is created by executing all possible paths, is highly computationally expensive. In [117] it is proposed to alleviate this problem by using JPF to generate models of agent programs that can be executed by other model-checkers.

This idea is further developed in [63], which shows how AJPF can be modified to generate models in the model languages of Spin or Prism. JPF uses backtracking points to explore the entire execution space of a Java program, in AJPF this process is used to track and number all states of the agent and to construct a symbolic model within the Java virtual machine. The LTL model checking algorithm is then executed on this symbolic model. When converting to Prism, the authors use a modified version of AJPF that uses a new class to deal with the probabilistic aspect of the model. Probability distributions are defined with instances of this class and the Prism model is then generated by looking at all the numbered states generated with JPF and transition probabilities found with the new class. This method brings along a few drawbacks: the first problem is that even though the symbolic model is generated directly from the agent code, it still requires a significant computational effort as the program explores the entire execution space of a Java representation of the agent code. The second problem comes with practicality and accessibility: probability distributions are defined in the Prism program by modifying the new Java class presented in the article, adding an additional step to the development process as the programmer has to modify the methods of a Java class in addition to developing the agent logic in a AOP language. These problems are addressed with the implementation proposed in this thesis by allowing the programmer to include probability distributions directly into the agent code and then automatically generate a complete Prism program for verification by model checking.

2.2.1. Verification by model checking

The problem with axiomatic verification is that proofs are not always simple, and it is hard to generalise a technique for large classes of systems. Even systems that share similar architectures can perform with substantially different logic. This is the main reason for the wider use of semantic techniques, such as model checking [13, 52], over axiomatic techniques for the purpose of verification, especially for agent-based systems. The first probabilistic model checkers were proposed in the 1980s and 90s [55, 56, 208], however the first industrial strength algorithms were developed in the early 2000s [58, 102].

The verification by model checking process, first proposed in [51], relies on the close relationship between models for temporal logic and finite-state machines. Assuming that a generic program $\mathcal{P}$ needs to be verified against a specification ψ, the process can be summarised in two steps:

1. From $\mathcal{P}$ generate a model $\mathcal{M}_{\mathcal{P}}$ that captures all possible states and computations of $\mathcal{P}$.

2. Determine whether or not the specification ψ is valid on $\mathcal{M}_{\mathcal{P}}$. If the result is positive, then the program $\mathcal{P}$ satisfies the specification ψ.

In some applications it is possible to generate a model that is tailored to the given specification so to reduce the size of the model, but often the model is independent from the specifications. In the latter case once the model is built, different specifications can be run without having to rebuild the model. Most software used to perform the verification on a model, for a give specification, will generate a so called *counterexample* [98], that is the first sequence of states and transitions (trace) found in the state space that does not satisfy the specification in question.

This theory can be in principle applied to any system, and therefore to any agent-based system. The main problem lies in the fact that even assuming that the actual model-checking (step 2) can be easily performed for any model, step 1 remains non-trivial: given the wide variety of architectures and AOP languages, the modelling process is very hard to automate. Even when focusing on a single architecture and a single language, different agent programs can generate widely different models. This thesis describes a method to alleviate the problem, with an automatic modelling technique that includes probabilistic modelling of the environment, and applies to a specific, but still widely applicable, set of agent-based systems.

2.2.2. Formal definitions

As highlighted in Subsection 2.2.1 the first step for applying verification by model checking is to construct a model of the system. The model of a system is designed to capture important properties of the system in order to reproduce its behaviour as completely as possible. For this particular application only discrete models will be considered, as the goal is to represent the agent reasoning which operates in loops (reasoning cycles) and a full transition to a new state only happens at the end of each loop.

The description of a system at any given time is given by a "snapshot" that captures the value of all the significant variables the model is supposed to consider: the *state* [52]. The model is then complete when it also describes how the system *transitions* from state to state. A common way to represent such a model is with a *state transition graph* or *Kripke Structure* [116]. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state [52]. These structures can be extended to include *probabilistic* behaviour so to model, for instance, unpredictable behaviour, environmental uncertainties and so on. This is done for example by specifying the probability of the system making a transition from one state to the other. There are many probabilistic models available that can be used for the purpose of verification by model checking, but this application is focused on two in particular: Discrete-Time Markov Chains (DTMCs) [168] and Markov Decision Processes (MDPs) [77].

A DTMC is a Kripke structure that allows for the definition of probabilities for the transitions of the system. For each transition a probability value is defined that describes the probability of it to take place. A formal definition of DTMC is as follows [134].

Definition 2.3 (Discrete-Time Markov Chain (DTMC))**.** *Given a fixed, finite set*

B of atomic prepositions, a (labelled) DTMC is a tuple

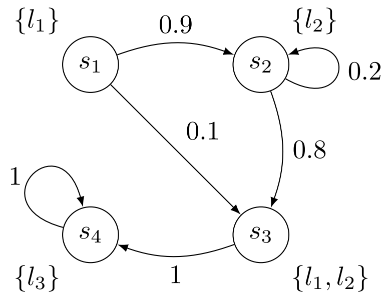
$$\mathcal{D} = \{S, s_0, \boldsymbol{P}, \mathcal{L}\}$$

where

- *S is a countable set of states.*
- *$s_0 \in S$ is the initial state.*
- *$\boldsymbol{P} : S \times S \to [0, 1]$ is a* transition probability matrix *where $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$.*
- *$\mathcal{L} : S \to \wp(B)$ is a labelling function that assigns to each state $s \in S$ a set of atomic prepositions $\mathcal{L}(s)$ from B that are valid in the state.* □

In a DTMC each element $\boldsymbol{P}(s, s')$ of the matrix $\boldsymbol{P}$ is the probability of a transition from state s to state s' to take place. Note that the condition $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$ implies that no deadlocks are allowed in this model, therefore all terminating states have self-loops with probability 1. Another important property of DTMCs, and Markov Chains in general, is that the conditional probability of future states does not depend upon the sequence that leads to the present state. A simple example of DTMC is represented in Figure 2.3. In DTMC and similar models there is no notion of *real-time*, however it is possible to keep track of the number of transitions as discrete time-steps. In order to include the notion of real-time one would need to use probabilistic models such as Probabilistic Timed Automata (PTA).

The second step of verification by model checking is to determine whether or not a given specification holds true in the model. For DTMCs the specification language Probabilistic Computation Tree Logic (PCTL) [99] can be used, which is an extension of Computation Tree Logic (CTL), and it captures probabilistic relationships between states, and the likelihood of *paths* to happen in run-time,



$$\boldsymbol{P} = \begin{pmatrix} 0 & 0.9 & 0.1 & 0 \\ 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.3.: *An example of DTMC with its transition probability matrix. Circles indicate states, arrows indicate transitions, numbers on arrows indicate the probability of the transition to take place. Each state can be labelled with a set of labels.*

where a path is simply an ordered sequence of states and transitions. Here follows a formal definition of PCTL.

Definition 2.4 (Syntax of PCTL)**.**

$$\begin{aligned} \phi &::= \ \texttt{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \texttt{P}_{\bowtie p}[\psi] \\ \psi &::= \ \texttt{X}\ \phi \mid \phi\ \texttt{U}^{\leq k}\ \phi \end{aligned}$$

where a is an atomic proposition, $\bowtie \in \{\leq, <, >, \geq\}$ and $p \in [0,1]$. $\texttt{X}$ is the 'next' operator, and $\texttt{U}^{\leq k}$ is the 'bounded until' operator with $k \in \mathbb{N} \cup \{\infty\}$. □

Definition 2.4 shows a distinction between two types of specifications: *state formulae* ϕ, which are evaluated over states, and *path formulae*, which are evaluated over paths and are generally used only as parameters for state formulae. For instance a state s satisfies a state formula $\texttt{P}_{\bowtie p}[\psi]$ if the probability of a path happening from s satisfying the path formula ψ, lies within the range specified by '$\bowtie p$'. Usual abbreviations are also allowed such as '$\texttt{F}\ \phi$' ('eventually', equivalent to '$\texttt{true}\ \texttt{U}\ \phi$'). A *reward* (or *cost*) is an association of a state or transition to a numerical value. A *reward structure* is defined as a pair (r_s, r_t) of *state reward* function $r_s : S \to \mathbb{R}_{\geq 0}$ and a *transition reward function* $r_t : S \times S \to \mathbb{R}_{\geq 0}$. PCTL formulas can be extended with *reward* properties [134] by the addition of the *reward operator* $\texttt{R}_{\bowtie r}[\cdot]$ and the following state formulas:

$$\mathtt{R}_{\bowtie r}[\mathtt{C}^{\leq k}] \mid \mathtt{R}_{\bowtie r}[\mathtt{I}^{=k}] \mid \mathtt{R}_{\bowtie r}[\mathtt{F}\,\phi] \tag{2.3}$$

where $r \in \mathbb{R}$, $k \in \mathbb{N}$ and ϕ is a PCTL state formula. Intuitively $\mathtt{R}_{\bowtie r}[\mathtt{C}^{\leq k}]$ is true if the expected reward *cumulated* before step k lies within the bounds expressed by $\bowtie r$, $\mathtt{R}_{\bowtie r}[\mathtt{I}^{=k}]$ is true if the reward at time step k meets the bounds expressed by $\bowtie r$ and $\mathtt{R}_{\bowtie r}[\mathtt{F}\,\phi]$ is true if the expected cumulated reward before a state satisfying ϕ occurs is within the bounds expressed by $\bowtie r$. Reward properties allow to represent richer specifications with the addition of quantities that are related to the temporal evolution of the system, for example a state reward may describe that in one particular state the system consumes a certain amount of power, and a specification for the model checker can be set to verify what is the expected power consumption (*cumulative reward*) within a fixed number of time steps.

The second model that is considered here is Markov Decision Process (MDP). The purpose of modelling with MDP is to generalise DTMCs with the addition of *nondeterminism.* In general this allows to model how a controller might actively make decisions on the system, where these decisions are not probabilistic in nature, e.g. the decisions cannot be described with a probability distribution. A formal definition of MDP is as follows [136].

Definition 2.5 (Markov Decision Process (MDP))**.** *Given a fixed, finite set B of atomic prepositions, a (labelled) MDP is a tuple*

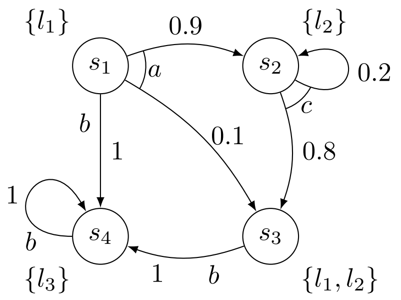
$$\mathcal{M} = \{S, s_0, C, \mathrm{Step}, \mathcal{L}\}$$

where

- *S is a countable set of states.*

- $s_0 \in S$ *is the initial state.*
- C *is an alphabet of choices with* $C(s)$ *being the set of choices available in any state* s.
- $\mathrm{Step} : S \times C \to \mathrm{Dist}(S)$ *is a probabilistic transition function with* $\mathrm{Dist}(S)$ *being the set of all probability distributions over* S.
- $\mathcal{L} : S \to \wp(B)$ *is a labelling function that assigns to each state* $s \in S$ *a set of atomic prepositions* $\mathcal{L}(s)$ *from* B *that are valid in the state.* $\square$

In each and every state s several choices $C(s) = \{a \in C \mid \mathit{Step}(s,a) \text{ is defined}\}$ may be available, at least one to avoid deadlocks, each corresponding to a probability distribution over other states. At each step an action from $C(s)$ is chosen *nondeterministically*. Secondly, a state is selected randomly, according to the associated probability distribution $\mathit{Step}(s,a)$. A resolution of nondeterminism in a path of a MDP is called an *adversary* of the MDP. In other words an adversary is responsible for choosing an action in each state of a MDP. This implies that once an adversary is applied, a MDP reduces to a DTMC, otherwise called the *induced DTMC* for the MDP. A simple graphical example of MDP is shown in Figure 2.4.



$$\mathit{Step} : \begin{array}{c} \left(\begin{array}{cccc} 0 & 0.9 & 0.1 & 0 \\ \hdashline 0 & 0 & 0 & 1 \\ 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{array}\right) \end{array} \begin{array}{c} a \\ \hdashline b \\ c \\ b \\ b \end{array}$$

Figure 2.4.: *An example of MDP with its transition function represented in matrix form. Circles represent states, arrows represent transitions. Numbers on transitions represent the probability of the transition to take place, lowercase letters represent nondeterministic choices. Each state can be labelled with a finite set of labels.*

Similarly to DTMCs, specifications for MDPs can be written with PCTL [30], however the semantic must be applied to all adversaries of the MDP, therefore to all induced DTMCs. In this scenario the $\mathtt{P}_{\bowtie}[\psi]$ reduces to the calculation of *minimum* or *maximum* probabilities over the full range of adversaries for the MDP model. The following form will be used to describe the minimum/maximum probabilities operator.

$$\mathtt{P}_{\mathtt{min}=?}[\psi] \mid \mathtt{P}_{\mathtt{max}=?}[\psi] \tag{2.4}$$

where ψ is a PCTL path formula.

Although formal definitions of DTMC, MDP and specification language syntax are reported in full for better clarity in later chapters, a detailed description of the methods and techniques used to verify DTMCs and MDPs goes beyond the scope of this work. An excellent guide on the topic can be found in [190] and in tutorial papers [82, 134, 136].

2.2.3. PRISM

The verification software package used in this work is PRISM [107, 135, 205]. PRISM is a probabilistic model checker, developed primarily at the Universities of Birmingham and Oxford, that allows to verify a variety of different probabilistic models, such as Discrete-Time Markov Chains (DTMCs), Markov Decision Processes (MDPs), Continuous Time Markov Chains (CTMCs) and PTAs.

Probabilistic models in PRISM are described using a proprietary state-based language. The model is then compiled to symbolic data structures based on BDDs (Binary Decision Diagrams) [43] and MTBDDs (Multi-Terminal Binary Decision Diagrams) [87], which allow to define models with significantly large state spaces with a reduced amount of memory, and which are fast to search and access.

Probabilistic properties in the PRISM language can be expressed in one of the following property specification languages: PCTL (Probabilistic computation tree logic, CSL (Continuous Stochastic logic), LTL (Linear Temporal Logic) and PCTL*, an evolution of PCTL that inherits some properties from LTL.

One of the great advantages of PRISM is the possibility of organising the description of the model into so called *modules*. Modules are independent entities within the model that operate on specific variables. Modules can access and read values of variables defined in other modules, but only each module is allowed to modify its own variables. This is particularly important with this application because it allows to synchronise different parts of the system so to make some operations execute in parallel. This concept will be more clear in 4.

Another feature which is useful for the scope of this work is the possibility in PRISM to define *reward structures*, which allow to analyse characteristics of a system that are correlated to particular states, for example a bump in energy consumption correlated to a system being used at full power.

2.3. Skills and algorithms for autonomy

The agent reasoning of an agent-based system usually works on a high level of abstraction by mapping abstracted sensory information to *function calls* that represent actions for lower level subsystems to execute. These lower level of abstraction subsystems, also known as *skills*, are part of the agent architecture and they are usually developed in a language that is different from the agent program. Potentially any algorithm or operation can be implemented as a skill of the agent, from sensing to motion planning to actuation [194].

The approaches to the implementation of the LISA system described in this thesis are intrinsically modular so that any skill that the agent might need during its operation can be implemented in some way.

2.3.1. Sensing and filtering

At the lowest level of abstraction, skills interface with the environment by commanding actuators and filtering sensed data. In a mobile robotic application this could be for example waypoint following skills, which would use sensed data to adjust motors speed so to make the robot head towards the next waypoint, or filtering and data fusion skills [9, 71, 72, 96, 217], so to infer particular values such as position, distance from a point and so on, by processing large amounts of data. For example filtering can be used to predict the trajectory of an external vehicle [48]. An excellent guide on the topic of filtering for mobile robots can be found in [199]. All autonomous mobile robots that operate in unknown environments use these algorithms and theories for localisation and mapping in one way or another, so Simultaneous Localization And Mapping (SLAM) [68, 73, 74, 162] skills are a common addition in those instances.

2.3.2. Motion planning

A common problem that recurs all over robotics is that of motion planning. Originally formulated as the *piano movers problem* [185, 192], motion planning refers to the process of determining, given a known environment, how to gradually move a body from an initial placement to a goal placement, while avoiding collision with obstacles [49, 138, 140–142]. The problem is obviously not restricted to the simple path planning in a linear reference frame: in most cases the planning has to be computed with rigid body transformations that can be applied to a rigid body representing the robot. This augmented state space is called the *configuration space* or *C-space* [150]. The part of the configuration space where the robot is allowed to move, e.g. where there are no obstacles, is called *free space* or *free configuration space*. It is important to notice that sometimes it is necessary to take into account the dynamics of the robot, increasing the complexity of the problem in terms of computational effort required to generate free paths. The computational complexity also increases when modelling more Degrees Of Freedom (DOFs) as the size of the search space grows exponentially with the number of DOFs.

Even though the motion planning problem sits in the continuous configuration space, the computation of feasible paths is usually discrete. Over the last few decades the motion planning problem has been tackled in a variety of different ways, however the most popular implementations can mainly be grouped under two schools of thought: *combinatorial* planning and *sampling-based* planning.

Combinatorial planning [95, 191] consists of characterising the configuration space by capturing all necessary information to perform planning from any starting point to any goal point. Most combinatorial methods first compute a *roadmap*: a graph that contains a list of points in the free configuration space and 'simple' paths that connect pairs of points through the free space, calculated according to predefined criteria such as minimum clearance from obstacles, shortest path and so on. Once the graph is constructed, the motion planning problem reduces to a simple graph search. A few examples of combinatorial path planning algorithms can be found in [121–123, 126]. Even though combinatorial motion planning algorithms are *complete*, in the sense that they are *guaranteed* to find a path where there is one, they are usually very difficult to apply where there is non-linearity, often the case for mobile robots models. Even more so when the environment is dynamic, in which case time can be included as part of the state space [133].

Quite differently from combinatorial algorithms, sampling-based motion planning algorithms [124] avoid the explicit characterisation of the configuration space by incrementally probing the free space. For each sample they use a collision detection method to check whether or not the new sample and the path to it fall within the free configuration space. Once the goal point has been reached the algorithms reconstructs a path back amongst the points that have been probed and possibly smooths out the path as much as possible to increase efficiency. The way the *probing* of the 'unknown' space happens varies amongst different algorithms. Some of them are based on heuristic methods and treat the problem as a graph search such as A* [66, 100] and more modern evolutions of it [45, 78, 128, 145, 197]. Other algorithms probe the environment more or less randomly, for example Rapidly-exploring Random Tree (RRT) [139] and newer implementations of it [125, 151, 218].

In [119], a framework is proposed to choose amongst multiple available path planning algorithms, based on an assessment of environmental complexity.

2.3.3. Combined motion and task planning

Apart from combinatorial and sampling-based path planning algorithms there is yet another approach to the motion planning problem that is worth mentioning, sometimes referred to as *symbolic approach* or simply *discrete planning.* The idea is to discretise the continuous configuration space and then use formal methods to generate discrete plans based on given LTL specifications [173]. Initially proposed as a way to generate discrete plans over a discrete decision space [12, 27, 120], it was then extended to generate continuous trajectories for mobile robots while still satisfying temporal logic formulas [50, 75, 175, 201], and later the same principles were applied considering dynamical models [76, 174].

The main process of the discrete planning can be summarised in three steps [75]:

1. *Discrete abstraction.* The robot configuration space is abstracted to a finite set of equivalence classes, for example with cellular [54] or triangular [21] decomposition. This results in a discrete set of prepositions that are associated with the decomposed subsets of the continuous world. In most cases this process requires the satisfaction of the bisimulation property [4] in order to ensure that the satisfaction of LTL [177] specifications holds for both the discrete and the continuous model. In this way any sequence of states in the discrete model is

associated to one trajectory in the continuous model.

2. *Discrete planning.* Using formal methods, and in particular model checking, a sequence of discrete states that satisfy a temporal logic specification is generated.

3. *Controller synthesis.* A control strategy is implemented at the continuous level while preserving the satisfaction of the temporal logic formula [200]. An example in [148].

The application of this concept can be found extensively in the literature, especially in the HS community. The fact that it is possible to give specifications in LTL allows to use this method to implement algorithms that "guide" lower level path planning algorithms such as RRT, and to implement complex instructions instead of point to point motion planning. LTL uses atomic prepositions operated with traditional logic ($\wedge$ and, $\vee$ or, $\neg$ not) and with with temporal operators ($\bigcirc$ next, $\Diamond$ eventually, $\cup$ until, $\Box$ always) [1]. For example the following expression represents the task of eventually visiting all of the areas in a subset of a map, in no particular order:

$$\Diamond p_{A_1} \wedge \Diamond p_{A_2} \wedge \cdots \wedge \Diamond p_{A_n} \tag{2.5}$$

where p_{A_i} represents the preposition 'area A_i explored'. Similarly for an example of partial ordering:

$$(\neg p_{A_3}) \cup ((p_{A_1} \vee p_{A_2}) \wedge \bigcirc p_{A_3}) \tag{2.6}$$

which represents the task of exploring area A_3 only after area A_1 or A_2 have already been explored.

An interesting application of this can be found in [130, 131] which uses the al-

[1]Note that although different symbols are used here to be consistent with the conventions of the scientific community, the temporal operators have the same meaning as the ones used for PCTL in Definition 2.4.

gorithm in [172] to generate an automaton that satisfies an LTL specification and it then uses the controllers described in [54] to integrate the automaton in a overall hybrid controller that satisfies the specification. In [47, 127] hierarchical abstractions (multiple layers) are used to control swarm robotic systems. The work presented in [221] uses optimal control to minimise a weighted average cost function to produce optimal trajectories that satisfy LTL specifications.

Recent applications [28, 29, 158, 159, 209] have been used to integrate the discrete planning process with sampling-based motion planning algorithms. This area of research defines hybrid spaces consisting of discrete and continuous components. The continuous layer is used to model high-dimensional robotic systems with non-linear dynamics and sampling-based algorithms are used to search for feasible paths. The discrete abstraction is used to simplify and optimise the tree search that arises from the sampling-based algorithms while satisfying complex specifications usually expressed with LTL. An exception to the trend of using LTL for specifications can be found in [137] where controllers are synthesised from PCTL specifications to account for uncertainties that can be modelled with probability distributions.

Other examples of symbolic planning can be found in [20, 86], where in order to ensure the satisfaction of the bisimulation property these approaches use so called *motion primitives*, a collection of dynamically feasible motion behaviours that the robot can execute while still maintaining satisfaction of temporal logic specifications.

Although discrete planning has been used in several instances to implement advanced motion planning in real applications, it is still limited by the complexity that arises when modelling high-dimensional robotic systems with complex high-level reasoning. Another problem not yet addressed by these methods is that even if full dynamics of the system are taken into account, and the bisimulation property holds, at some point the physical robot may not be able to execute the control strategy generated by the algorithm, for example it may be faced with unmapped obstacles or some actuators may be faulty.

The approach towards autonomy taken with discrete planning algorithms is different from that of agent-based systems as they tend to unify planning and reasoning in a single framework, penalising ease of implementation and modularity. One of the great advantages of modern agent-based systems is in fact the possibility of programming them with NLP languages and the possibility of integrating them with potentially any kind of algorithm as skill. However the integration of these approaches themselves as skills in an agent-based system can potentially improve the planning capabilities of the agents, as they offer a more advanced and flexible approach to the problem of motion planning, compared to classical sampling-based or combinatorial algorithms. The possibility of defining complex temporal logic specifications translates into actions that implement more complex behaviour in an agent-based system, potentially reducing complexity of the agent reasoning and in turn of the agent program. An agent-based architecture which implements reactive behaviours can also address the need of replanning when the robot is faced with unpredicted problems by interrupting the current action and re-call the algorithm to generate a new trajectory with the latest available sensory information.

Chapter 3.

The Limited Instruction Set Agent

A novel architecture for autonomous control, called Limited Instruction Set Agent (LISA), is presented in this chapter. The aim of this new implementation is to provide programmers with a unified framework to describe both the agent reasoning and a model of the environment, so that a software algorithm can automatically generate a verifiable model, which can accurately describe agent reasoning within its environment.

3.1. Introduction

The Limited Instruction Set Agent (LISA) is a novel agent-based system that offers a unified framework to model and verify the reasoning process of BDI-based agent systems.

The architecture of the LISA system is structured in a layered way, with a BDI-based agent reasoning on top and a set of skills with lower and lower levels of abstraction, where skills are subsystems dedicated to the execution of specific tasks, generally run and coordinated by agent reasoning. The agent reasoning operates with symbolic information, literals that represent abstract concepts, rather than with numeric measurements. The translation of numerical data into symbolic information, and vice versa, is performed by the skills of the agent. From the prospective of the agent reasoning, skills are actions that are executed through *action calls*, commands that invoke skills associated with predicates that represent actions in the agent program. Some of the skills are initialised at the beginning of the operation of the agent and executed in a continuous fashion. This is generally the case for lower levels of abstraction, for example skills that interface sensors or skills that monitor energy consumption in a mobile platform. In most cases these skills do not share data directly with the agent reasoning, but information is translated into perception predicates, Boolean variables that the agent reasoning uses to reason about future actions. The architecture of the LISA system is described in detail in Section 3.2.

The agent program is developed as an evolution of Jason [37]. Despite being a popular and effective software package for AOP, Jason was not developed with automatic verification in mind, and some of its features and characteristic make automatic modelling for verification by model checking extremely difficult. To improve this aspect, the LISA system features an agent reasoning process that is designed to facilitate modelling, so to automatically generate discrete models that can be easily verified with any probabilistic model checker. The agent program is implemented using a NLP software package called sEnglish [211], which uses natural language sentences in an easily readable document so that a human operator can understand the reasoning process of the agent, without the need for advanced programming training. In the LISA system the sEnglish language is enhanced with structures that allow to define probabilistic models for environmental variables, so that a single document can provide enough information to generate a complete and verifiable model.

3.2. The agent architecture

The architecture of an agent-based system describes how the agent logic communicates with lower abstraction subsystems to gather information that it uses to make decisions, and how actions calls invoke subsystems skills and ultimately operate in, and influence the environment. The research community has proposed many agent-based architectures, many of which share the characteristic of being structured in a layered way, with skills at the bottom that directly communicate with the environment, and increasingly higher levels of abstraction up to the agent reasoning.

Although the principles behind the LISA system could be applied to several different architectures, this BDI implementation is based on the *three-layer* architecture [88], with communication between the middle layer to form *abstract loops* [163]. The BDI-based reasoning operates on conceptualised abstract structures called *literals* that can represent different aspects of the environment, of the internal state of the agent itself or represent actions that the agent executes by activating skills.

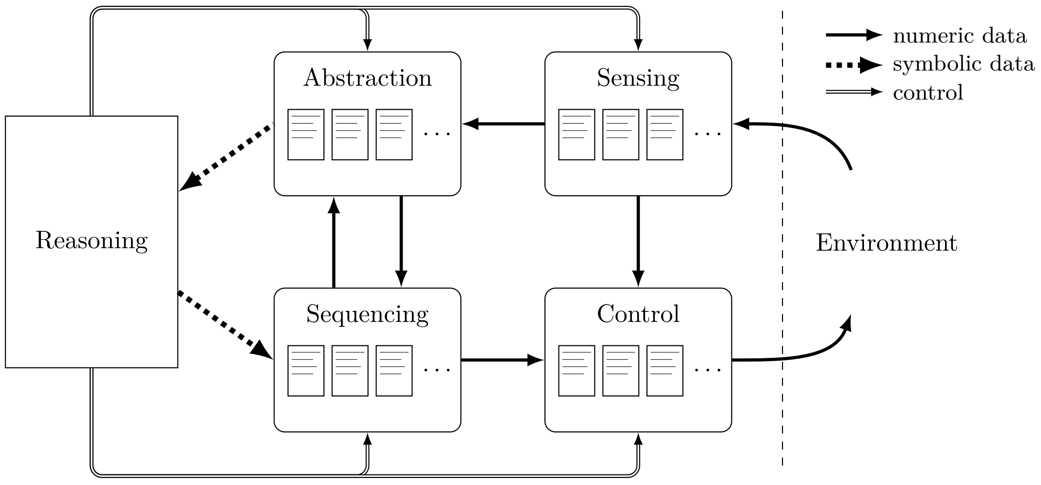


Figure 3.1.: *The LISA architecture. Labelled boxes with rounded corners represent categories of skills which can interact with each other or with other skills. The agent reasoning activates and controls each skill. The environment is anything external to the agent, for example sensors and actuators.*

Figure 3.1 shows a schematic representation of the architecture. The first observation to be made is that this represents in fact a *hybrid system*. There are two types of data flow: *symbolic* and *numeric*. Symbolic data is discrete and generally with much lower granularity than discretised sensing signals. Symbols, or literals, are the basic form of data on which the agent reasoning operates. Numeric data flows can be discrete as a result of discretisation from sensing equipment, but from an abstract point of view they can be considered continuous. It is also important to observe that the agent reasoning has universal control access to every subsystem, in other words it is responsible for activating (and deactivating) each and every skill.

Skills are classified under two broad categories: *single execution* and *continuous execution*. Single execution skills are usually associated to processes that do not require constant monitoring, and they generate a feedback that is translated to the agent by other dedicated skills when the execution is terminated. Continuous execution skills are associated to processes that the agent reasoning is generally not designed to closely monitor, i.e. lower levels of abstraction. Skills that are designed to be active throughout the full execution of the agent have to be still activated: the agent program offers the possibility of defining a set of initial actions that can be used for the purpose.

In order to clarify the purpose of each group of skills, here follows a brief description of the functionality and operation of each subsystem.

Sensing. Information on the state of the environment at any given time comes from physical sensors. Any sensor is susceptible to some level of noise. The Sensing skills are responsible for translating streams of noisy data coming from sensors into data that other skills can deal with. In most cases the process consists of filtering algorithms associated with routines that organise the data into standardised structures. An example of a Sensing skill could be for instance a data fusion algorithm that computes the position of a mobile robot relative to a map of the environment, based on data coming from several sources. Another class of skills that falls under the classification of sensing is that of communication skills. In the LISA system, incoming messages are treated as percepts and passed on to Abstraction skills to verify their validity and ultimately deliver them to the agent reasoning. Sensing skills can also implement mechanisms to prioritise or select incoming messages according to the specifications of a particular implementation.

Abstraction. The agent reasoning is designed to make decisions upon a state of the world described by a set of available symbolic predicates, rather than a set of numerical variables. The Abstraction skills function as an interface between agent reasoning and Sensing skills, they take care of translating a stream of input data into a pre-defined set of Boolean variables that the reasoning is able to deal with. For example for a mobile robot, an Abstraction skill would be one that looks at a target position, calculates the distance from the target and produces symbolic statements such as '`I am near destination`' or '`I am at destination`'. Abstraction skills are also responsible for translating incoming messages from external agents or operators into activation or deactivation commands for beliefs to be added or removed from the Current Beliefs set.

Sequencing. Once the agent reasoning has made its deliberations it will issue action commands to the Sequencing skills. The Sequencing skills take care of translating a command that is expressed in symbolic form into a sequence of lower level actions or into a numerical instruction for the lower level skills to operate. The communication with Abstraction skills allows access to the most updated state of the world and to communicate back intentions and updates for the agent reasoning. For example if the agent reasoning of a mobile robot issues a command such as '`Go to point X`', Sequencing skills would retrieve the value of the variable `X`, then retrieve an updated map from the abstraction or Sensing skills, then execute a *path planning* algorithm to generate a sequence of safe waypoints, then finally pass this information on to navigation skills.

Control. The Control skills represent the interface of the agent with physical actuators and, more generally, hardware that influences the world in some way, for example motors or communication devices. The closed loop with the Sensing skills allows for automatic control and monitoring of basic operations such as movement or communication. An example for a mobile robot would be a '`waypoint following`' skill that makes sure that the robot is heading towards the next waypoint defined by the path planning algorithm, or a '`send message`' skill that for example sends pre-defined messages and monitors receipt acknowledgments from Sensing skills.

The architecture described in this section is general enough to be applicable to large variety of agent-based system. This structure drives the philosophy behind the agent reasoning of LISA and in turn many traits of the implementation will reflect it. However the agent reasoning of LISA can in fact be modified to work with different BDI-based architectures, by tweaking the top section of the agent program to accommodate the initialisation of different subsystems, and adjusting the action definitions.

3.3. The agent reasoning

At the core of the decision-making of every agent there is the agent program, which maps the information available to the agent to actions that aim to influence the world to bring it to a state that is close enough to a goal state as per initial specifications [189]. The agent reasoning of the LISA system is based on the BDI paradigm and it is constructed as an evolution of Jason [37]. All modifications to the Jason structure are made with the purpose of addressing design traits that cause an increase in the size of the state space required to model the agent reasoning for verification by model checking.

The agent reasoning is operated in iterations called reasoning cycles. The reasoning cycle of LISA reasoning is presented here in mathematical terms with a step-by-step illustration, highlighting for each step what is the difference relative to Jason and the reasons behind each improvement.

The agent program of LISA systems is developed with the NLP language SYSTEM-ENGLISH (sENGLISH) [211] (see Section 2.1.2). Thanks to the Cognitive Agent Toolbox (CAT), sENGLISH programs can be compiled into Jason agents, therefore making the LISA system backwards compatible with Jason, with some restrictions due to the addition of some features that are described in detail throughout this section.

The ultimate goal of this implementation is to provide programmers with a framework that includes automatic finite-state modelling and verification of the agent, by allowing the inclusion of probabilistic data about the environment the agent will be placed in. The environmental data and responses are defined within the agent program itself. The agent program will then be automatically compiled into code for a probabilistic model checker, as explained in details in Chapter 4.

3.3.1. From Jason to LISA

The structure of the LISA reasoning is very similar to that of Jason. The most important difference lies with the fact that Jason allows for the handling of a single event and then for the execution of a single action per reasoning cycle, where the LISA system works in a multi-threaded way, avoiding the need for functions that are external to the agent program to select events and actions. This modification brings an advantage when generalising the process of modelling the agent reasoning straight from the agent program, as all relevant information can be included within the agent code.

This concept will be more clear after the step-by-step explanation of the reasoning cycle of Subsection 3.3.2, and with the mathematical proofs of Section 4.2. Here all the major changes from Jason to LISA are described, grouped under six main categories: perception, messages, beliefs, goals, logic rules and external actions.

General operation

The LISA reasoning implementation features a multi-threaded workflow, which simplifies the modelling process of agent reasoning and significantly reduces the number of states required to describe it. The operation of Jason relies on a set of functions that are external to the agent code and that implement some choices within the reasoning cycle, for example *message selection* and *event selection*. When developing a model of the agent reasoning, these functions have to be seen as "external" in the sense that they make seemingly arbitrary choices against the current state of the agent, making them a source of nondeterminism for the model. For example priority given to one event rather than another is completely dependant on the particular application, and can hardly be generalised if not for very special cases.

Avoiding the need for developing functions that are external to the agent code also gives the advantage that the full operation of the agent can be defined in a single document, with a single programming language. In the case of Jason for example, even though default versions are provided (usually simple First In, First Out (FIFO) queues) these functions are defined in Java and they are not easily accessible, creating the need for the developer to investigate and understand the underlying structure of the framework.

Perception

In LISA perception predicates can be of two types: *sensory perception* ($p \in B_s$) and *action feedbacks* ($p \in B_a$), therefore the Beliefs set is defined as:

$$B = \{B_s, B_a, B_m\} \tag{3.1}$$

where B_m is the set of all possible mental notes, beliefs that are activated by internal actions. The *action feedbacks* are percepts that actions generate in order to make the agent reasoning aware of their outcome, i.e. success, partial success or failure. For the purpose of modelling, this classification is very important: the different nature of sensory percepts and action feedbacks needs to be modelled in a different way for the model to accurately describe the behaviour of the environment.

In Jason, action feedbacks are also present to recognise when an action is completed, but not in the form of beliefs directly visible to the agent, they are implemented as Boolean variables returned by the method that performs the action call. Messages are treated as any other belief and each message can be considered either sensory percept or action feedback depending on how it is defined in the agent program.

Messages

A Jason agent features a message handling system for messages coming from external agents. In Jason the handling of the messages happens internally to the agent reasoning, as a step of the reasoning cycle. In particular messages are queued and one message for each reasoning cycle is handled, then the message goes through a 'social acceptance function' that verifies whether the agent can handle it or not. Furthermore messages themselves can directly generate events. Similarly for the LISA implementation, messages are treated as perception beliefs, and only messages that match a pre-existing database are accepted. However messages cannot generate events directly as the modification they bring to the Current Beliefs set is subject to the application of logic based implication rules, which in Jason does not happen until a later stage. This concept will be more clear after the step-by-step explanation of the reasoning cycle in Subsection 3.3.2.

Messages that are not part of the message database are all transformed to a special belief that will let the agent know that an unknown message is received, so that the user can define a plan that manages the situation. However this task is delegated to abstraction and Sensing skills, so that the agent reasoning only really receives messages that are part of the database. The reason behind this is that an infinite set of possible messages cannot be modelled as part of a finite-state machine, as it would undermine the applicability of the modelling methods described in Chapter 4. There will also be a trade-off to be considered between the size of the message database and the size of the resulting finite-state model.

This modification allows to avoid the modelling of the messages handling within the model of the agent logic, therefore greatly reducing the number of states required for the final model of agent reasoning.

Beliefs

Jason, as well as similar languages, makes extensive use of *first-order logic* to combine multiple atomic predicates into more complex beliefs. In particular every belief can be accompanied by round brackets that contain an *object*, which gives a context to the initial atomic predicate. For example '`tall(tree)`' expresses a particular property - that of being *tall* - of the object *tree*. In sENGLISH this concept is improved even further by associating objects to variables. The variable type becomes part of the sentence and it must be specified in the ontology file. For example in the sentence '`Go to location L`' the object '`L`' is of type '`location`' which may be specified as a vector of coordinates, a set of Euler angles, and so on. This information can then be used to coordinate skills, for example to apply different inputs to the same skill.

Another common feature of languages that are derived from AgentSpeak is that there is an additional degree of abstraction over beliefs: they can be true or false but also present or not present in the Current Beliefs set. The agent can believe that a predicate is true or *not* true (the true version of it is not in the Current Beliefs set) but also that the predicate is false or *not* false (the false version of it is not in the Current Beliefs set), giving an additional degree of control over *triggering events*. Although this feature is present in Jason, and in turn in sEnglish, it is dropped in the LISA system - at this stage of development - for beliefs that are simply Boolean variables, in favour of ease of modelling for verification purposes. The consequence of this is that a belief that is not present in the Current Beliefs set has the same meaning as a belief that is false, and to achieve the same level of abstraction the programmer has to define additional beliefs to indicate different states of a particular concept.

Goals

In Jason and similar BDI agents there is a distinction between the concept of belief and the concept of goal. Ideally beliefs represent what the agent knows about the world, and goals represent a state that the agent would like to achieve. In a practical sense this distinction does not have a great influence: beliefs and goals can both trigger plans, with the only difference being that goals are automatically removed from the Current Beliefs set once the plan is completed. The only practical advantage that goals can have in the agent program is that when the addition of a goal is part of a plan it can be used to trigger and include a different plan into the current one. This can be achieved by simply taking care of copying the required actions in place of the goal in question. For these reasons in the LISA system the definition of goal is dropped in this first stage of development, by implementing goals as mental notes. As for the modifications on beliefs, this simplifies the syntax and therefore the process of generating a model directly from the agent code.

Logic rules

In Jason logic-based implication rules are present but yet not very well implemented, to the point that the main text itself [37] advises against their use. Rules, in AgentSpeak derived languages, can only be used as context for plan triggering and do not constitute a way to apply modifications to the Current Beliefs set in any way. Logic rules are a great tool when implementing a BDI agent, as they allow to implement advanced and complex reasoning in a schematic way, making them a valuable addition to the sequential nature of plans. For this reason, in the LISA system, rules are implemented so that they can make changes to the Current Beliefs set and therefore generate events.

This feature does not directly simplify the modelling or the verification process, but it allows to reduce the number and length of plans required to reproduce the desired logic, and therefore reduce the state space needed to model it.

External Actions

The LISA reasoning implementation introduces a new classification for external actions that can be either of type `runOnce` or `runRepeated`. This reflects the dual classification of single execution or continuous execution mentioned in Section 3.2. As the name suggests `runOnce` actions terminate themselves after a single execution; `runRepeated` actions on the other hand, activate routines that require the agent to actively stop their execution with a `stopRepeated` command. Both types send action feedbacks to the agent in the form of beliefs. Even though this feature does not simplify the modelling process, it gives the user a greater level of flexibility for the implementation of rational behaviour. For example `runRepeated` actions can be used to activate continuous processes such as perception processes or waypoint following processes.

For the purpose of modelling and verification, there will be no distinctions between `runOnce` and `runRepeated` actions as long as the programmer takes care of properly defining the action feedbacks for them, and `stopRepeated` commands are considered as independent actions in the discrete model.

3.3.2. The LISA reasoning cycle

With reference to Definition 2.1, the reasoning cycle of a LISA $\mathcal{R}$ can be summarised with the following 5 steps. The reasoning cycle of LISA systems is based on the reasoning cycle of Jason, and it is presented here with direct comparisons to the latter.

Figure 3.2 shows a schematic representation of the reasoning cycle of the agent. The numbering of the functions blocks (rounded corners and diamond shapes) reflects each step of the reasoning cycle. Note that all the steps described here are all part of a single time step t that is the integer count of reasoning cycles throughout the agent operation.

Step 1: Current Beliefs update

The first step of the reasoning cycle is to update the Current Beliefs set $B[t]$ with the most recent available data. This operation is done by a function called Belief Update Function (BUF), denoted with f_{BU} in Figure 3.2, which updates the Current Beliefs set from its previous version $B[t-1]$ to a new version $B[t]$. The BUF takes as input a set of beliefs paired with instructions on what to do with each belief, namely add them or delete them from the Current Beliefs set. The input comes from two sources:

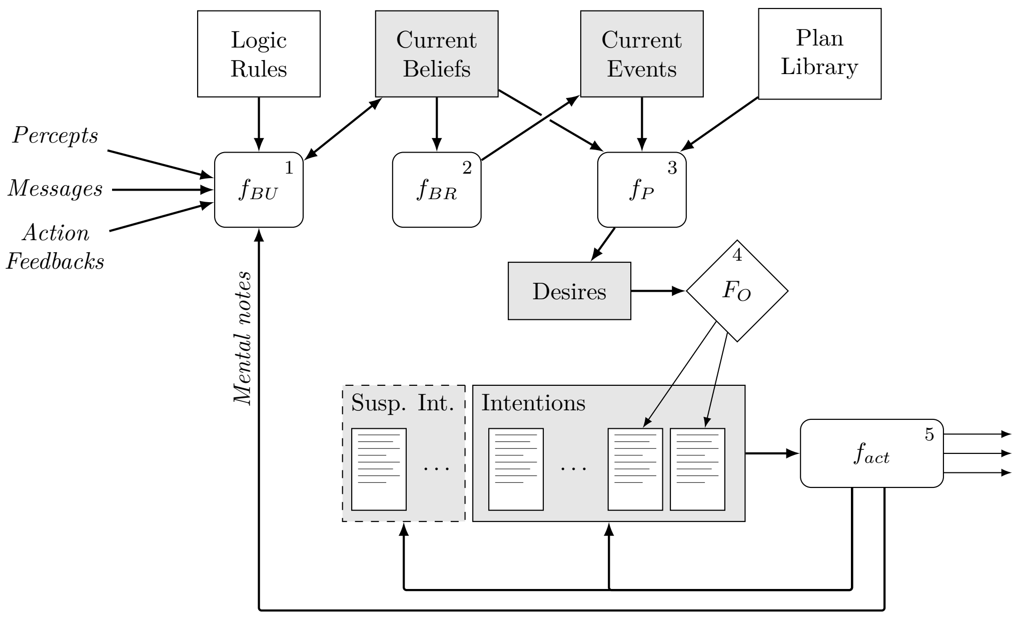


Figure 3.2.: *The LISA reasoning cycle. Blocks with rounded corners represent internal functions, diamond-shaped blocks represent external functions, white square blocks represent static sets, grey blocks represent dynamic sets. Functions are numbered according to the order of execution within the reasoning cycle.*

- *Abstraction skills* generate beliefs and instructions coming from sensory perception, incoming messages and action feedbacks of external actions.
- *Internal actions* generate mental notes.

Additionally, the BUF looks at the database of logic based implication rules and applies all the necessary modifications to the Current Beliefs set $B[t]$. Generally speaking, if there is a conflict between a mental note update issued by an internal action and a logic based implication rule, the latter is given priority and the Current Beliefs set updated accordingly. This represents a strong difference from Jason, where logic based implication rules do not represent a way to modify the Current Beliefs set, and they are only applied in Step 3 when verifying the applicability of a plan.

Another difference lies in the fact that, in Jason, incoming messages are handled internally to the agent reasoning, so there are additional steps in the reasoning cycle, that precede the Current Beliefs update, for the message checking, message selection (only one message per reasoning cycle is allowed) and acceptance check. In particular a dedicated function called *Message Selection Function* F_M, which is external to the agent program, selects the message that will be dealt with in the current reasoning cycle. In the LISA system message handling is delegated to skills and messages are treated as any other perception belief, so there is no need for a dedicated function and processes for the agent reasoning to handle them.

Step 2: Current Beliefs review

The update of the Current Beliefs set generates events that will in turn trigger plans for the agent to execute. Events are beliefs that are copied from the Current Beliefs set $B[t]$ to the Current Events set $E[t]$ paired with an operator from $\{+, -\}$ that indicates that the belief has been added or deleted. This operation is done by a function called Belief Review Function (BRF), denoted with f_{BR} in Figure 3.2, which maps $B[t-1]$ and $B[t]$ to a new Current Events set $E[t]$.

Events can also be classified as either *internal* or *external*. External events are those associated with changes in perception beliefs, action feedbacks and messages ($p \in \{B_s, B_a\}$) and internal events are those generated from internal actions, e.g. addition or deletion of mental notes ($p \in B_m$). In Jason there is an additional distinction for internal events as they can be either changes in mental notes or changes in goals. This is not the case for the LISA system as goals are also implemented as mental notes (see Subsection 3.3.1). In Jason, events can also be directly generated by internal actions, which is not the case for the LISA reasoning as logic rules are applied before events are generated (see Step 1).

In Jason only one event per reasoning cycle is dealt with. This implies that the Current Events set in Jason functions as a queue. For each reasoning cycle the BRF pushes new events in the queue, and then, in a following step, a function called

Event Selection function

$$F_E : \wp(E) \rightarrow E[t] \tag{3.2}$$

selects a single event from the currently available ones and passes it on for plan handling and execution. The multi-threaded workflow of the LISA system allows to deal with every event that is available for each reasoning cycle, hence the Current Events set is empty at the start of the reasoning cycle. The reason for this is that avoiding the implementation of functions such as the Event Selection Function greatly simplifies the abstraction process for model checking, as explained in details in Section 4.2.

Step 3: Retrieving applicable plans

Now that the set of current events is in place, the agent needs to make decisions on what to do with each event. Each plan of Π is a sequence $\pi_j(\lambda_j)$, with $\lambda \in [0, n_{\lambda_j}]$, where the first element $\pi(0)$ is called *triggering condition.* In AgentSpeak-derived languages the latter is usually expressed in the form

$$e_j \;:\; c_j \;\leftarrow \tag{3.3}$$

where $e_j \in E$ is an event called the *triggering event* and c_j is a logic condition on beliefs from B called the *context.* When an event in $E[t]$ matches the triggering event[1] of a plan $\pi_j \in \Pi$ the plan is *triggered.* However this is not enough for the agent to decide to commit to executing said plan. For each triggered plan the agent checks whether or not $B[t]$ satisfies the context ($B[t] \vDash c_j$). In Jason, logic based implication rules are applied at this stage rather than during the Current Beliefs set update.

All the plans that are triggered by an event and $B[t]$ satisfies their context are copied

[1]A different expression for 'matching' is that the triggering event can be *unified* with the event [37].

into a subset of the *Desires* set $D[t]$. Once this operation is performed on every event of $E[t]$, the latter is reset to the empty set and the Desires set becomes:

$$D[t] = \{D_1[t], D_2[t], \ldots, D_{n_e}[t]\} \tag{3.4}$$

where each $D_j[t]$ is the set of plans triggered by an event $e_j \in E[t]$ and $n_e = |E[t]|$ is the number of events at time t. The function that performs these operations is internal to the agent (in the sense that it is embedded in the framework) and it is indicated with f_P in Figure 3.2. In Jason only one event per reasoning cycle is selected, therefore the Desires set contains only plans that are triggered by the event that was selected in the previous step.

Step 4: Plan selection

Once all the applicable plans are copied into the Desires set, if more than one plan was found to be applicable for any event, the agent has to make a choice on which plan to pursue for that particular event. This operation is performed by a function that is external to the agent program, called *Applicable Plan* (or *Option*) *Selection Function*

$$F_O : \wp(\Pi) \rightarrow \Pi \tag{3.5}$$

that maps a set of plans to a single plan from the plan library. Since the Desires set is composed by groups of plans relative to different events, the Plan Selection function must be applied to each of them. This is clearly not the case for Jason where a single event is selected for each reasoning cycle. When defining a discrete model of the agent reasoning, being external to the agent reasoning, the Plan Selection function has to be considered as a nondeterministic entity. From the prospective of the agent program, all applicable plans are by definition equally applicable, with no particular priority amongst them, therefore the Plan Selection function represents an arbitrary decision and it must be modelled as such. This issue is addressed in Section 4.2 where the abstraction process is described including the nondeterministic choice and a special case where nondeterminism can be avoided. In Section 4.5 a new method is described to use the automatically generated model to implement an advanced Plan Selection function that automatically adapts to each particular agent.

The final result of the operation is a set of plans called *intentions* that are copied into the Intentions Set $I[t]$. It is important to note that plans are *copied* into the Desire set from the Plan library, but not exclusively, which implies that different subsets of $D[t]$ may have a copy of the same plan. However, if a plan is selected multiple times in the same reasoning cycle, it will only be copied once in the Intensions set. Furthermore once a plan is copied to the Intentions set for execution, if the plan is selected again it will not be copied in the Intensions set a second time, but it will carry on from the current state (index) unless a plan interruption action is issued.

Step 5: Actions execution

Once a plan is part of the Intentions set, the agent is committed to execute it. The final step of the reasoning cycle is the execution of actions from intended plans. At the end of every reasoning cycle the agent takes the next available action from each plan and it calls an external function from a skill, if the action is external, or passes instructions to the BUF for mental notes to be added or removed from the Current Beliefs set in the next reasoning cycle.

Once an action is issued, it is removed from the plan in question. The function that performs these operations is indicated with f_{act} in Figure 3.2.

Plans are sequential lists of actions, therefore if an action is not completed the plan cannot carry on. Internal actions are executed within a single reasoning cycle, hence the agent assumes their execution to be instantaneous. The way that the agent is made aware of completion of external actions is through *action feedbacks*. If the last executed action from a plan has not returned an action feedback yet the plan is held into a special substructure of the Intentions set called *Suspended Intentions*. This operation is managed automatically by the system and the developer is not required to implement any method that takes care of it.

In Jason, at every reasoning cycle, the agent is only able to execute a single action. For this reason an external function called *Intention Selection function*

$$F_I : \wp(\Pi) \rightarrow \Pi \tag{3.6}$$

selects the plan to be executed in the current reasoning cycle. This is again an additional source of nondeterminism for a model of agent reasoning, and one of the reasons the LISA reasoning is implemented as a multi-threaded workflow is to avoid the need to include F_I in the model.

It becomes clear at this stage that in order to characterise the behaviour of the agent the point to which the agent has advanced each plan must be part of the *state* of the model along with the Current Beliefs set. This concept is better explored in Section 4.2.

To summarise, the LISA reasoning cycle was based on the Jason reasoning cycle but a few yet fundamental modifications make its operation substantially different. The main difference is that LISA reasoning is operated in a multi-threaded way that allows to eliminate some of the functions that are external to the agent code in the Jason implementation: the *Message Selection* function (F_M), the *Event Selection* function (F_E) and the *Intention Selection* function (F_I). These three functions do not make a significant impact on the execution of the agent as they pick one item at a time from a pool of items rather then choosing one of them and discarding the rest. In other words F_M, F_E and F_I only determine the order of consideration of the respective sets they are selecting from. There is however another function that is external to the agent program, which still remains with the LISA implementation: the *Plan* (or *Option*) *Selection* function. The reason for this choice is that this function has a significant impact on the operation of the agent as it will choose one of the plans from the Desires set and it will discard the rest, therefore potentially changing the outcome of the mission.

3.4. The agent program

The agent program is the body of code that is used to describe and implement a desired logic for the agent. There is a variety of different AOP languages (see Section 2.1.2) in most cases dedicated to the implementation of a specific subclass of autonomous agent. The LISA agent program is based on sEnglish with some additional features that allow the programmer to include discrete probabilistic models of the environment along the definition of environmental variables. The ultimate goal is to produce an agent program that is compilable to a discrete model for automatic verification, in this case into the input language of the popular model checker Prism [135]. One of the benefits of using sEnglish is that it uses NLP syntax, so it is easily readable and it does not require extensive training to be used in a productive way.

sEnglish projects are structured as follows. There are three types of files: a main *reasoning* file (extension `.sej`) is used to describe the main logic of the agent, an *ontology* file (extension `.ont`) allows to define hierarchies of variable types that the agent uses to coordinate skills, and a group of *action* files (extension `.sep`), one for each external action used in the main file, which describe the way the action command is operated and the associated skill is invoked.

The syntax of sEnglish is rather minimal and can be mainly summarised with the following points

- Square brackets '`[...]`' enclose natural language sentences.
- If preceded by a 'hat' symbol '`^[...]`' the sentence is a *belief* (percept or mental note). Any sentence that is a belief can be negated with a tilde '~' symbol.
- '+' and '−' symbols signify *addition* or *deletion* of the belief they precede from the Current Beliefs set. The same syntax is used for expressing both *events* and internal actions.
- Sentences that are enclosed by square brackets but not preceded by any symbol represent *external actions*. Each external action must be associated with an action `.sep` file. External actions can only be listed as part of plans or within the list of initial actions.
- With minor exceptions (namely the triggering condition of plans), each action or sentence is ended with a dot.
- Common keywords are used to articulate logic statements, such as `and`, `or`, `not`, `while` and so on.

As mentioned in Section 2.1.2, the main program is structured similarly to Jason, and in fact the *sEnglish Publisher* - an Eclipse plugin for sEnglish - includes a compiler for Jason. The LISA implementation does not change the structure of the sEnglish reasoning file itself, which is structured in sections as follows. Sections are titled with all capital letters in the reasoning file.

1. `INITIAL BELIEFS AND GOALS.`
   This is where the programmer reports a list of all the beliefs of $B_0 \subset B$, which are copied into $B[t]$ at the beginning of the first reasoning cycle. Beliefs in

this section can be expressed without being encapsulated in square brackets, as long as they are separated on different lines and each line ands with a dot.

2. `INITIAL ACTIONS.`

   All the actions in $A_0 \subset A$ listed here are executed before the first reasoning cycle takes place. Initial actions can also be expressed without square brackets.

3. `PERCEPTION PROCESS.`

   In this section it is possible to configure objects for world modelling and Boolean symbolic sentences that are in turn used within the document to represent perception inputs. In LISA all the percepts must be listed in this section, except for action feedbacks which can be listed in the dedicated action '`.sep`' files.

4. `REASONING.`

   A list of all logic based implication rules L that are applied to $B[t]$ for every reasoning cycle. In the LISA system, rules can add or remove beliefs from the Current Beliefs set. Rules are expressed in the form:

   ```
   If <condition> then <action>
   ```

   where `<condition>` can be any logic based rule on beliefs from B and `<action>` is an internal action of addition or deletion of a mental note from $B[t]$.

5. `EXECUTABLE PLANS.`

   The Plan Library Π. Each plan is listed in the form:

   ```
   If <triggering_event> while <context> then
     <action>
     <action>
     ...
     <action>.
   ```

where the first line is the *triggering condition* and `<action>` can be either external, if they call external functions through their action files, or internal, if the add or delete beliefs from the Current Beliefs set. For a practical example of a sEnglish plan see Figure 3.3.

```
1 //Plan 5
2 If ^[Block explored] while ^[Areas left unexplored] and ~^[Sea state
     is too high] then
3   [Activate park mode.]
4   [Generate set of waypoints Wi.]
5   +^[Re_exploring areas]
6   [Activate drive mode.].
```

Figure 3.3.: *Brief example of plan definition in a sEnglish document. Line 2 is the* triggering condition*, lines 3-6 are external and internal actions.*

Action definition files are used to describe the way the agent is going to issue action executions, e.g. invoke skills. Each `.sep` file lists a set of simple characteristics which include:

- *Procedure name*, which has to match the file name.
- *sEnglish sentences* the action is associated to.
- *Process, repeat mode*: the subsystem in which the action is implemented in (see Figure 3.1) and the type of action (`runOnce` or `runRepeated`)
- *Input and output classes*: if the skill needs inputs or outputs they must be defined here.
- *sEnglish code*: the action can be defined as a sequence of sub-actions represented by other sEnglish sentences
- *Performance feedback*: the list of all possible action feedbacks for the action.

Procedure name, *sEnglish sentences* and *Process, repeat mode* are mandatory fields. Additional fields are allowed for specific applications, for example *matlab libraries* and *matlab url* when implementing the action as a Matlab function. It is also possible to organise actions in sections and include a section number as part of the ‘sep’ file.

In the main reasoning file, actions are invoked by using one of the sentences of the section *sEnglish sentences*. In case of runRepeated actions, a stopRepeated action can be issued by adding the prefix ‘Stop’ before the action sentence in the reasoning file.

Figure 3.4 shows an example of the syntax for a position calculation action, note that the ‘Execute’ command in the ‘senglish code’ field is used by the sEnglish Publisher to generate Matlab scripts.

```
procedure name:: computing desired position and state for fixed
    attitude
senglish sentences:: Compute desired position Pdes and desired state
     vector Xdes for fixed attitude.
mol reference::
input classes and local names::
output classes and local names:: position [Pdes], desired state
    vector[Xdes]
senglish code::  Execute "␣Pdes␣=␣circular_trajectory_position2
    ([150,120],70,tLim,t,circlePhase);
QuatDes=[0;0;0;1];␣Xdes=[Pdes;zeros(3,1);QuatDes;zeros(3,1)];␣".
matlab libraries::
conceptual graphs::
author data::
section number:: 2
performance feedback::
```

Figure 3.4.: *Example of action definition ‘sep’ file in sEnglish, which contains all necessary information to impelement the particular function.*

3.4.1. Probabilistic modelling of the environment

In order to generate a probabilistic model of the environment directly from the agent code, the latter has to feature the probability distributions that describe the probabilistic nature of some parts of the system. This Subsection gives an overview of how the sEnglish language is enhanced, in the LISA system, with the possibility of defining, within the agent code, probability distributions to describe the environment.

The way in which probability distributions of random variables are found widely depends on the particular application. For example distributions of environmental variables are usually inferred from large amount of data collected with physical sensors or through simulation (a recent example in [109]), while rate of failure of actuators and sensors are usually provided by the manufacturer.

In the LISA system framework there are only two sources of probabilistic behaviour, and they both come in the form of perception predicates: *sensory percepts* and *action feedbacks*, which include messages. Beliefs in the LISA system are simply Boolean variables, so the uncertainty comes with the amount of time that passes between changes in the state of the variable. In particular for sensory percepts this time can be completely random, unless there is a known underlying probability distribution to describe its behaviour. Action feedbacks on the other hand show a different kind of probabilistic behaviour: first of all they can only be activated when the related action is actually invoked, and also whether they carry a message of failure or success, they are guaranteed to eventually come true, at least within a set time limit that generates a failure message. It is important to remember that in this setting time indicates the integer counts of reasoning cycles, which can be arbitrarily spaced and not necessarily equally spaced in real time.

The approach taken here to model both the probabilistic behaviour of perception beliefs and action feedbacks is to include in the agent code a finite set of key values that characterises a pre-defined probability distribution. The particular distributions chosen for the purpose, explained in the following subsections, give no particular advantage over other distributions, and they are shown here as a proof of concept rather than a result. In fact this principle can be extended to any other distribution that can be completely characterised with a finite set of numerical values.

A different approach could be for example to give a discrete, bigger, set of values that characterises the full distribution at each consecutive time step.

Action feedbacks

Action feedbacks are percepts that action execution functions return to the agent reasoning after the action has been executed. In the LISA framework the programmer has to take care of defining all possible action feedbacks and their probability distributions. The way this is done is by including within the action definition file (`.sep`) for each action all information necessary to characterise the probability distribution.

Figure 3.5 depicts the distribution of choice for the characterisation of the probabilistic nature of action feedbacks activation. The probability of an action feedback of becoming true has a value of zero at the time of the action call and it increases linearly from a minimum value of $1/(2\sigma + 1)$ when $t = \mu - \sigma$, where σ is a *variance* value, up to 1 when $t = \mu + \sigma$. This is in line with the fact that action feedbacks are bound to become true after a set period of time. At each time step the probability of activation increases of a factor of $1/(2\sigma + 1)$.

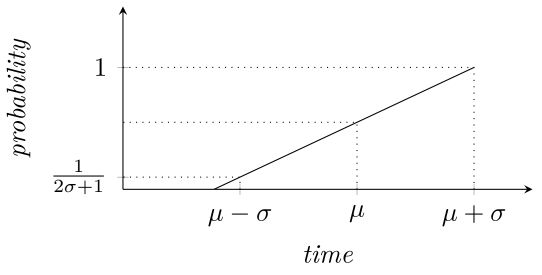


Figure 3.5.: *Probability distribution* $\Pr_{af}[t]$ *for a single action feedback activation. μ is the average time of activation and σ is the variance. In case of multiple action feedbacks, the probability at each time is divided proportionally amongst each action feedback according to pre-defined weighting factors.*

The probability distribution of Figure 3.5 can be formally expressed as:

$$\mathrm{Pr}_{af}[t] = \begin{cases} \dfrac{t-(\mu-\sigma)+1}{2\sigma+1} & |t-\mu| \leq \sigma+1 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

Note that since at each time step t there is a probability $\mathrm{Pr}_{af}[t]$ that the action feedback is activated, there is clearly a $1 - \mathrm{Pr}_{af}[t]$ that it does not happen. This creates self-loops in the discrete model and allows the system to be modelled as a Discrete-Time Markov Chain (DTMC) or Markov Decision Process (MDP), as both models do not allow for deadlocks (see Definitions 2.3 and 2.5).

When multiple action feedbacks are defined for a single action, the model accounts for it by splitting the probability amongst them, as a weighted sum:

$$\begin{gathered} \mathrm{Pr}_{af}[t] = p_1 \cdot \mathrm{Pr}_{af}[t] + p_2 \cdot \mathrm{Pr}_{af}[t] + \cdots + p_{n_{af}}\, \mathrm{Pr}_{af}[t] \\ \text{with} \quad p_1 + p_2 + \cdots + p_{n_{af}} = 1 \end{gathered} \tag{3.8}$$

It becomes clear from Equations 3.7 and 3.8 that this distribution can be completely characterised by including in the action definitions of the agent program a set of three values for each available action feedback:

1. *A probability value* p, which represents the weighting factor in case multiple action feedbacks for the same action are present.

2. *The average number of reasoning cycles* μ in which every action feedback is expected to become true.

3. *The variance* σ around the average number of reasoning cycles.

The triad of values is embedded along each action feedback enclosed with square brackets and comma separated. Note that the value of μ and σ must be equal for all action feedbacks of an action, in fact different values specified for the other action

Table 3.1.: *Example of probability values over time for the two action feedbacks of the example in Figure 3.6. Each time step t represents a full reasoning cycle.*

t	1	2	3	4	5	6	7	8
$\Pr_{\texttt{continue}}[t]$	0	0	$0.6 \cdot \frac{1}{5}$	$0.6 \cdot \frac{2}{5}$	$0.6 \cdot \frac{3}{5}$	$0.6 \cdot \frac{4}{5}$	0.6	0
$\Pr_{\texttt{Abort}}[t]$	0	0	$0.4 \cdot \frac{1}{5}$	$0.4 \cdot \frac{2}{5}$	$0.4 \cdot \frac{3}{5}$	$0.4 \cdot \frac{4}{5}$	0.4	0

feedbacks are ignored. Figure 3.6 shows an example of the syntax described above from an action '`.sep`' file.

```
procedure name ::  wait instructions
senglish sentences ::  Wait for instructions.
process, repeat mode :: control, runOnce
...
...
performance feedback :: Continue[0.6,5,2], Abort[0.4,5,2]
```

Figure 3.6.: *Partial example of a '`.sep`' file action definition with action feedbacks with probabilistic modelling (line 6) in sEnglish. The numbers in square brackets characterise the probability distribution depicted in Figure 3.5.*

Once activated, action feedbacks are detected by the agent reasoning within one reasoning cycle, given the multi-threaded implementation of the LISA system, therefore the variable associated with them is deactivated after one reasoning cycle.

Table 3.1 shows what are the probability values generated for the example given in Figure 3.6, assuming that the action '`Wait for instructions`' is executed at time $t = 0$.

Sensory percepts

Generally speaking sensory percepts present a behaviour that is less predictable than action feedbacks as they are not guaranteed to come true within finite time intervals, therefore needing a probability distribution for modelling activation and one for modelling deactivation. In order to be able to describe their probabilistic nature over time with a finite number of numerical values, the approach taken here is to define a single probability distribution that is symmetric around an average, with a given variance, and evenly space copies of the same distribution over time by a given amount, for example the average value. Another difference from action feedbacks is that a sensory percept does not necessarily have to be deactivated after a fixed amount of reasoning cycle. This phenomenon needs to be accounted for with a second probability distribution for deactivation. As discussed in the introduction to this section, the choice of distribution made for this application is not driven by any particular advantage if not that of being easily implementable, but it is still applicable to a large variety of phenomenons. A reasonable alternative could be for example a discrete Gaussian distribution, which could be drawn with the same amount of information.

An additional degree of modelling is given in the LISA system by allowing the user to define *conditionality* for each percept. For example a percept of the kind '`i am at destination`' cannot possibly be activated when the robot is not at destination but it is also not moving. The way this is implemented is that any percept p_b defined as conditional to a percept p_a, will only have a chance to be activated if p_a is active for at least an amount of time equal to the average value for p_b has passed, and if p_a becomes inactive during this time, the associated counter is reset. This concept will be better explained in Section 4.3.

Figure 3.7 shows the distribution of choice (triangular distribution) for the characterisation of the probabilistic nature of sensory percepts. Similarly to action feedbacks the distribution is characterised by a probability value p, an average value μ and a variance σ. The distribution is then repeated at time intervals equally spaced by a value of μ. The same distribution is used here for both activation and deactivation of the sensory percepts variables. Each distribution of Figure 3.7 can be

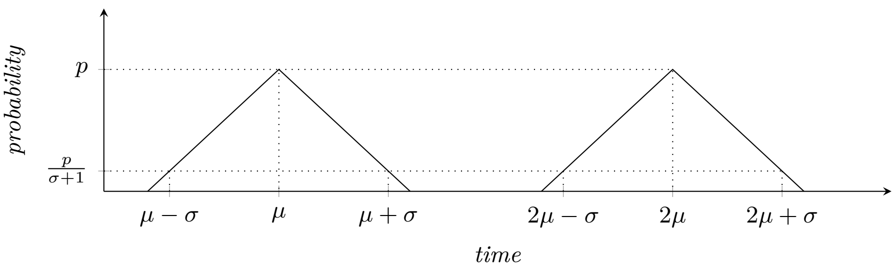


Figure 3.7.: *Probability distribution* $\mathrm{Pr}_s[t]$ *for a sensory percept.* μ *is the average time of activation or deactivation,* σ *is the variance and* p *is the maximum probability value.*

formally expressed as:

$$\mathrm{Pr}_s[t] = \begin{cases} p_s\left(1 - \left|\dfrac{t-\mu}{\sigma+1}\right|\right) & |t-\mu| \leq \sigma+1 \\ 0 & \text{otherwise} \end{cases} \qquad \text{with} \quad p_s \in [0,1] \tag{3.9}$$

Analogue to the action feedbacks, at any time t there is always a $1 - \mathrm{Pr}_s[t]$ chance that the activation/deactivation does not happen.

The information needed to model the probabilistic nature of sensory percept is included in the reasoning file of the agent code under the '`PERCEPTION PROCESS`' section. Here the programmer must list all possible sensory percepts and their modelled probability values. Up to three set of values are needed for each percept:

1. *A list of percepts or mental notes* (optional) to which the percept being modelled is conditional to.
2. Probability, average number of reasoning cycles and variance of *activation.*
3. Probability, average number of reasoning cycles and variance of *deactivation.*

Each set of values is, as usual, enclosed in square brackets and the three are in turn enclosed in curly brackets. Figure 3.8 shows a simple example of a '`PERCEPTION PROCESS`' section with probabilistic modelling in sEnglish. Table 3.2 shows a simple

Table 3.2.: *Example of probability values over time for a percept modelled with $p = 0.6$, $\mu = 5$ and $\sigma = 2$. Each time step t represents a full reasoning cycle.*

t	1	2	3	4	5	6	7	8
$\mathrm{Pr}_s[t]$	0	0	$0.6 \cdot \frac{1}{3}$	$0.6 \cdot \frac{2}{3}$	0.6	$0.6 \cdot \frac{2}{3}$	$0.6 \cdot \frac{1}{3}$	0

example of probability values for activation or deactivation of a percept modelled as described above with no conditional percepts and values of `[0.6,5,2]`.

```
PERCEPTION PROCESS
Monitor the following booleans:
//Percepts
Sea state is too high. {[], [0.01,10,5], [0.01,10,5]}
I am at global waypoint. {[], [0.5,5,0], [1,1,0]}
Last waypoint reached. {[I am at global waypoint],[0.1,1,0],[1,1,0]}
```

Figure 3.8.: *Example of percepts with probabilistic modelling in sEnglish. The numbers in square brackets characterise the probability distribution depicted in Figure 3.7.*

Rewards

The last verification-oriented feature introduced with the LISA system is the possibility for the programmer to describe *reward* structures, that then allow to use reward properties as defined in Equation 2.3.

The reward values can be declared by listing '`name=value`' of each reward, comma separated and enclosed in curly brackets, on the same line of any percept declaration within the Percept Process section, or any action within any of the executable plans. Figure 3.9 shows an example of reward declaration of an action within a plan. In this case, for example, if analysing the cumulative '`fuel`' reward, each time that the internal action of adding a belief '`+^[Re_exploring areas]`' is performed the value of fuel goes up by a factor of 1. The model checker can then provide with an estimate of the total reward accumulated for example within a certain amount of time steps.

```
//Plan 5
If ^[Block explored] while ^[Areas left unexplored] and ~^[Sea state
     is too high] then
  [Activate park mode.]
  [Generate set of waypoints.]
  +^[Re_exploring areas] {fuel=1,time=1}
  [Activate drive mode.].
```

Figure 3.9.: *Minimal example of reward declaration for an action in a sEnglish plan.*

3.5. Conclusions

A new agent-based architecture is presented here, with a verification-oriented approach to allow for automatic modelling and verification using model checking software. The purpose of an autonomous system is to influence the environment in order to make it be closer and closer to the design specifications.

The LISA system is based on the BDI paradigm and features a layered architecture with the agent reasoning on top and lower level algorithms and control systems on lower levels of abstraction. Agent reasoning is the part of the agent that takes decisions on what actions to take in order to achieve its tasks. Each subsystem of the architecture is a group of skills that largely operate on the same inputs and outputs. The way that the agent reasoning interfaces and influences the world to achieve its design objective is by issuing action commands to the skills that in turn convert the command to yet lower level commands or directly execute the action.

Agent reasoning is based on AgentSpeak/Jason and its features and reasoning cycle are described in comparison to Jason. The main innovation in the LISA system is the implementation of a multi-threaded workflow that avoids the need of several specific functions that would be a source of nondeterminism for the discrete model, ultimately reducing the state space required to model the agent behaviour. The agent program is based on the sEnglish language with a few improvements on the framework, so to allow the programmer to include probability distributions to describe the behaviour of perception beliefs and action feedbacks.

Including all the necessary information enables the system to generate a complete and verifiable model of the agent reasoning, as will be discussed in the following chapter.

Chapter 4.

Automatic verification of agent reasoning

Verification is a crucial step in the development process of an autonomous system, that aims to guarantee safety of the environment and the machine itself when operating in real-world applications. Verification is usually performed by model checking, which requires a model of the system to be verified, expressed in a language that is specific to the model checking software of choice. This chapter shows that the LISA system can be modelled as a DTMC or MDP and how the agent program can be automatically translated into code for the model-checker PRISM. *Model checking techniques are also considered for predicting possible consequences of actions, so that the agent can select the best strategy.*

4.1. Introduction

THIS chapter describes how the agent reasoning of the Limited Instruction Set Agent (LISA) system can be automatically translated into a complete discrete model and verified with model checking software. Although this principle could be applied with any model-checker, this project focuses on PRISM [205].

The LISA system, and agent-based systems in general, are structured with an agent reasoning that deliberates on future actions, and a set of skills that perform said actions and gathers information from the environment. In this framework, skills can only be activated by agent reasoning, therefore verifying the agent reasoning represents a major step in proving the effectiveness and the safety of an agent-based robotic system.

In order to perform formal verification on agent reasoning, it has to be abstracted to a model that fully represents its behaviour. Such a model must include its interface with other subsystems and ultimately with the environment. In the case of the LISA system the interface is composed of symbolic data coming from Abstraction skills and action commands to execute sequences of actions that represent various physical and mental skills of the agent. Two finite-state models are considered for the abstraction of LISA reasoning: Discrete-Time Markov Chain (DTMC) and Markov Decision Process (MDP). Both models have discrete-time progression, which is in line with the fact that the agent reasoning operates in reasoning cycles and actions are only executed at the end of each of these cycles. DTMC models do not account for nondeterminism, and they will only be applicable in a special case. MDP models on the other hand are proven to be applicable to any implementation of LISA reasoning. Section 4.2 describes the abstraction process with proofs of applicability of both DTMC and MDP.

The inclusion of probabilistic information about the environment is introduced in Section 3.4.1 and it represents an important tool for the automatic generation of the discrete model. The probability distributions of environmental variables that are included within the code allow to generate complete models, in this case expressed in terms of PRISM code, that can be readily verified against probabilistic specifications, for example Probabilistic Computation Tree Logic (PCTL) specifications. The process of translation of the LISA agent program to a PRISM model is described in Section 4.3.

Once a model of the agent reasoning is generated, the verification process is straightforward. Model checkers such as Prism offer a great variety of tools that allow to deeply explore the properties of a system and verify that they meet design specifications. The verification process generates counterexamples that give the programmer knowledge about flaws in the agent program that can be iteratively corrected. The design-time verification process of the LISA system is described in Section 4.4.

Section 4.5 describes how the discrete model and verification tools offered by model checkers such as Prism can be used to improve agent programs, and in particular the LISA program. The availability of a complete probabilistic model of the system gives the possibility of making estimates on the outcome of actions in terms of probabilities, so that the agent can use this knowledge to opt for actions that bring it as close as possible to a desired world state.

4.2. Abstraction to finite-state machine

In order to perform verification by model checking of any system, a complete model of the system is needed. In case the system features probabilistic behaviours, verification by model checking can be performed on a probabilistic model. Advanced verification queries can be used to verify probabilistic specifications, for example PCTL specifications with the Prism model checker. A complete model of the system is one that completely describes its internal operation and its interaction with external entities, e.g. its interface with the world.

The operation of agent reasoning in a BDI agent-based system is intrinsically probabilistic: inputs come in the form of perception beliefs and even though precise patterns of activation may be known, generally speaking these are events that occur at time intervals in a probabilistic fashion and can therefore be described with probability distributions. One of the new features of the LISA system is that the user is able to include probability distributions of input variables within the agent code (see Section 3.4.1).

Even though model checkers such as Prism allow to verify a variety of models, this project focuses on two discrete probabilistic models in particular: DTMC and MDP. The reason for choosing only discrete time models is that the agent operates in reasoning cycles and the state of the reasoning is not relevant until the reasoning cycle is ended and beliefs, events and plan indices are all updated. This section aims to show that the agent reasoning of a LISA system can always be modelled as a MDP and in a special case it can also be modelled as a DTMC.

Figure 4.1 reformulates a diagram of the high level architecture of LISA, highlighting the interface between agent reasoning and its skills. Assuming that all necessary skills are implemented and associated to actions of the agent code, the only interface that the agent reasoning has with the outside is composed of two classes of inputs, percepts and action feedbacks in the form of beliefs, and one class of outputs, action commands. Note that in this framework messages from external agents are included as perception beliefs (see Section 3.3).

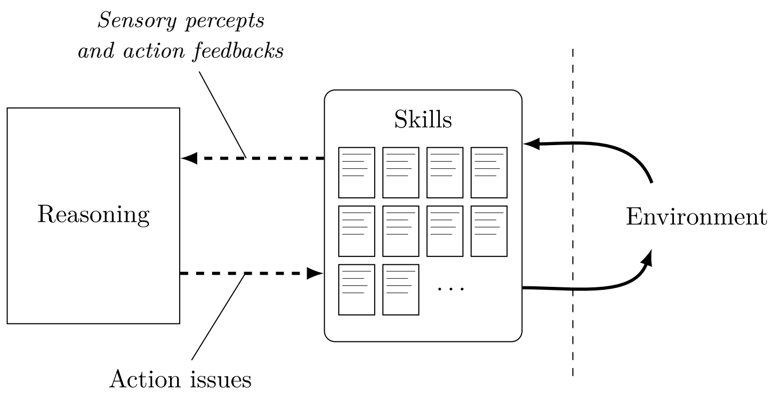


Figure 4.1.: *Agent reasoning interface with the skills. Information to the agent comes in the form of predicates representing sensory percepts or action feedbacks. The agent reasoning controls the skills through action commands, or action issues.*

A BDI agent of this type is completely defined, as per Definition 2.1, by listing all beliefs and actions, a set of rules and a set of plans that operate on those beliefs by sequentially executing actions. This is in principle a system with well defined states and transitions, assuming probability distributions of random inputs are known. However as shown in Section 3.3, with the Jason and LISA implementations there are some aspects of the reasoning cycle that introduce nondeterminism when modelling the agent reasoning.

In Jason the nondeterminism comes from four functions that are external to the agent reasoning listed here in temporal order of usage within the reasoning cycle:

- The *Message Selection* function (F_M) selects one out of the incoming messages for each reasoning cycle.
- The *Event Selection* function (F_E) selects one out of the events generated for each reasoning cycle.
- The *Plan Selection* or *Option Selection* function (F_O) selects one out of the set of Applicable plans (or Desires set).
- The *Intention Selection* function F_I selects one of the intentions from the Intentions set to be carried on in the current reasoning cycle

The nondeterminism of these functions comes from the fact that they are external to the agent code, therefore unless their operation is modelled along with the agent reasoning they represent in fact an arbitrary choice from the perspective of the agent code. Including a description of these functions in the model of the agent reasoning is possible but rather impractical: the functions are implemented externally to the agent code, usually in a different programming language, therefore making the modelling process significantly difficult to automate, and in most cases requiring the programmer to have a deep knowledge of the underlying structure of the agent program execution. For example in the case of Jason these functions are implemented in JAVA and to personalise them the user has to modify the default implementation within the custom JAVA libraries.

In the LISA system, thanks to its multi-threaded implementation, the nondeterminism is reduced to only one function: the Plan Selection function. The other functions listed above are used in Jason because of its single-threaded execution that allows to deal with a single message, a single event and a single intention per reasoning cycle. The Plan Selection function however is still present in the LISA implementation because it does not depend on the parallel execution and it plays an important role in the operation of the agent reasoning. The outcome of the Plan Selection function greatly influences the agent's behaviour.

Definition 2.1, among other things, introduces the concept of *plan* as a sequence $\pi(\lambda)$ with $\lambda \in [0, n_\lambda]$, where $\pi(0)$ is the triggering condition and $\pi(\lambda)$ with $\lambda > 0$ is an action from A. Assuming that a plan is not allowed to be executed multiple times in parallel, let us define a set of *plan indices*:

$$\boldsymbol{\lambda}[t] = \{\lambda_1, \lambda_2, \ldots, \lambda_{n_\pi}\} \tag{4.1}$$

where each index $\lambda_j \in \mathbb{N}_{\geq 0}$ represents the state of execution of a plan at time t. For instance, if a simple agent is programmed to have two plans, then a set of plan indices $\boldsymbol{\lambda}[3] = \{2, 4\}$ indicates that at time $t = 3$ the agent is currently executing action 2 of plan 1 and action 4 of plan 2. This information is clearly necessary to determine the current state of agent reasoning, as it will be shown in Theorem 4.1.

Depending on the particular application, plans will not generally have the same number of actions. For each plan π_j the range of values for each index is defined as a set $\boldsymbol{\Lambda}_j = \{0, \ldots, n_{\lambda_j}\}$ of natural numbers between 0 and the total number n_{λ_j} of actions. Consequently it is possible to define a set

$$\boldsymbol{\Lambda} = \{\boldsymbol{\Lambda}_1, \boldsymbol{\Lambda}_2, \ldots, \boldsymbol{\Lambda}_{n_\pi}\} \tag{4.2}$$

of all *possible indices* for all plans, where n_π is the total number of plans.

The following Theorem 4.1 shows a particular case where the LISA reasoning can be abstracted as a DTMC. This is potentially an advantage over using MDP as nondeterministic models can potentially be more computationally expensive to verify.

Theorem 4.1 (LISA reasoning abstraction to DTMC)**.** *Given a LISA $\mathcal{R}$, assuming the existence of sets of (discrete) probability distributions* $\mathrm{Dist}(B_s)$ *and* $\mathrm{Dist}(B_a)$*, over the set of percepts and the set of action feedbacks, if* $\forall\, i,j \in [1,n_\pi]$, $i \neq j$, $\pi_i(0) \neq \pi_j(0)$ *the LISA reasoning can be modelled as a DTMC .*

Proof. A DTMC is completely characterised given a countable set of states S and a transition function $\boldsymbol{P}: S \times S \to [0,1]$.

According to the definition of LISA, for a reasoning cycle to be completed the agent needs to be aware of $E[t]$, in order to recall plans from the plan library, of $B[t]$, in order to check the plans context, and of the state of the plans, in $I[t]$ in order to execute the next actions. The state of a LISA is only relevant at the end of each reasoning cycle, therefore a generic state at time t can be expressed as:

$$s[t] = \{B[t], E[t], \boldsymbol{\lambda}[t]\} \tag{4.3}$$

The set of Current Beliefs $B[t]$ and the set of Current Events $E[t]$ are subsets of finite countable sets (B and E respectively), they are therefore finite and countable. The set of current indices $\boldsymbol{\lambda}[t]$ for the plans of the agent is composed of values with a finite range, that is natural numbers between 0 and the length of each plan. The state space, given by

$$S = \wp(B) \times \wp(E) \times \boldsymbol{\Lambda} \tag{4.4}$$

is therefore finite and countable. Note that not all the states in the set S are reachable, only a subset of S will contain states that the agent can possibly reach during its execution. The state of the agent is initialised with $s_0 = \{B_0, \emptyset, \mathbf{0}\}$, where $\mathbf{0}$ is a null array of dimension n_π, and by triggering the actions listed in the set A_0 of initial actions.

The transition function of a DTMC describes the way in which the state of the model changes at every step. For each reasoning cycle actions from plans in the Intentions Set ($\lambda > 0$) are executed and the indices are updated. If the action was external, the index of the relative plan will only be updated if the associated action feedback is activated, e.g. the action feedback generated an event. Plans with $\lambda = 0$ can be triggered by new events. For each reasoning cycle, events can be generated from changes in beliefs, namely *mental notes*, *action feedbacks* and *percepts*. Changes in mental notes are given by internal actions, which are associated with the plan indices. Changes in action feedbacks and percepts are given by known probability distributions. If $\forall\, i, j,\ i \neq j,\ \pi_i(0) \neq \pi_j(0)$, e.g. if all plans have different triggering conditions, then

$$\forall t \in \mathbb{N}_{\geq 1}\ ,\quad \left| \bigcup_{k=1}^{n_e} D_k[t] \right| = |D[t]| \leq |E[t]| \tag{4.5}$$

the number of applicable plans is always less than or equal to the number of events, as each event will trigger at most one plan. This implies that the *Plan Selection* function F_O becomes a trivial one-to-one mapping, therefore the system does not show any nondeterministic behaviour.

Given that the state space of the LISA reasoning is finite and countable, the transitions between states can be defined with actions from plans and known probability distributions for action feedbacks and percepts, and there is no trace of nondeterminism, the modelling of the LISA reasoning with DTMC is complete. □

Theorem 4.1 proves that the LISA reasoning can be abstracted as a DTMC in the particular case when all plans have unique triggering conditions. Theorem 4.2 is now presented to prove that when this condition does not hold, the LISA reasoning can still be abstracted as a MDP.

Theorem 4.2 (LISA reasoning abstraction to MDP)**.** *Given a LISA $\mathcal{R}$, assuming the existence of sets of (discrete) probability distributions* $\mathrm{Dist}(B_s)$ *and* $\mathrm{Dist}(B_a)$*, over the set of percepts and the set of action feedbacks, the LISA reasoning can be modelled as a MDP.*

Proof. A MDP is completely described given a countable set of states S and a transition function $Step : S \times C \to \mathrm{Dist}(S)$, with $C(s')$ being the set of available choices in any state s'. As per Theorem 4.1 the set of states is given by $S = \wp(B) \times \wp(E) \times \mathbf{\Lambda}$ and it is finite and countable. The initial state is still defined as $s_0 = \{B_0, \emptyset, \mathbf{0}\}$. If $\forall\, i, j \in [1, n_\pi]$, $i \neq j$, $\pi_i(0) \neq \pi_j(0)$, according to Theorem 4.1, the system does not show any nondeterminism. However, if $\exists i, j \in [1, n_\pi]$, $i \neq j$: $\pi_i(0) = \pi_j(0)$, then at some time t'

$$\exists t' \in \mathbb{N}_{\geq 1} : \left| \bigcup_{k=1}^{n_e} D_k[t'] \right| > |E[t']| \tag{4.6}$$

the number of applicable plans is greater than the number of events, therefore for some event $e_k[t'] \in E[t]$ (with $k \in [1, n_e]$), the application of the Plan Selection function to the k-th subset of the Desires Set related to e_k involves a nondeterministic choice. Depending on the chosen plan, different future probabilistic outcomes from action feedbacks will be activated. The nondeterministic nature of the plan choice with F_O prevents the abstraction of the LISA reasoning with DTMC models. However, this choice represents the only nondeterministic part of the LISA reasoning, thus for each event $e_k \in E[t]$ the set of available choices is fully represented by $C_k(s') = D_k[t']$. Once a choice is made by the Plan Selection function for each of the available events, the transition to the next state is defined, as shown in Theorem 4.1, by looking at events generated by changes in mental notes, percepts and action feedbacks, and by updating plan indices accordingly. Given that the LISA reasoning features a finite and countable state space, a finite set of choices for each state and a well defined transition relation between states, it can be modelled as a MDP. □

Theorems 4.1 and 4.2 show that the LISA reasoning can generally be modelled as a MDP, and as a DTMC under the condition that all plans are implemented with different triggering conditions. Assuming that probability distributions of perception beliefs are known, the LISA reasoning can be abstracted as a MDP even if the Plan Selection function is unknown. This allows to verify the agent reasoning with a model checker by manually selecting the choices or even by exploring sets of chains of choices, e.g. sets of adversaries.

The possibility of defining agents that can be abstracted in two different ways brings along the question of what is the most efficient or effective way to do so. On one side DTMC models are generally less computationally expensive to verify, therefore defining agents that can be abstracted as DTMCs can potentially reduce the computational load on the model checker, making the verification process faster. However, in order to achieve similar behaviours, the programmer might be forced to define agents with larger sets of mental notes, longer plans and/or more logic based implication rules. This will make the state space larger[1] as well as requiring greater effort from the developer by limiting one of the features of BDI agent programming.

On the other hand MDP models can be more computationally expensive to verify, potentially making the verification process slower. However this is not necessarily the case, as allowing plans to be activated under the same conditions shift some of the load on the Plan Selection function, allowing in turn to define code with less beliefs and conditions and therefore generating models with a reduced state space. The MDP model does not include the choice itself in the transition function, but rather a set of choices that can be explored in a variety of ways. One thing that

[1]A larger state space generally implies larger memory required to store the model and more time for the model checker to explore the model when verifying queries.

is possible to do is, for example, to ask the model checker to find adversaries that satisfy specific conditions. This is a valuable tool that can be used to implement the Plan Selection function itself to explore the model for adversaries that minimise for example probability of failure, and then choose a plan based on the results given by the model checker. Assuming the correctness of the model checking software, in this way it is still possible to guarantee a completely verified process that also includes the Plan Selection function. This concept is explored in Section 4.5.

4.3. Generating PRISM models from agent code

The process of translating the agent reasoning code of the LISA system into a model in the input language of the probabilistic model checker Prism is described here, with detailed explanations of the modelling process and pseudo code of the end result in Prism. Once the Prism model is available, the software offers a wide range of tools for verification of properties expressed in PCTL, or in an evolution of it which includes reward properties (see Equation 2.3).

Section 3.4 described how the reasoning of the LISA system is implemented and an approach to a unified modelling solution that includes a probabilistic model of the world in which the agent is placed. This is done by using NLP with sEnglish to describe the logic and including key values within the sEnglish program to describe the probability distribution of activation/deactivation of percepts and action feedbacks over time. Additionally this framework allows to describe reward structures that can be used to verify reward properties. All of this gives enough information to automatically generate a DTMC or a MDP model in the input language of Prism.

The translation software was developed as a Matlab script. It is a simple text processing algorithm and it typically runs in the order of tens of milliseconds on the consumer laptop PC that was used for the testing. For this reason, and being a one-off execution for each system, the performances of the translator itself will be considered negligible for the results presented in the thesis.

In [63], Dennis *et al.* developed a process that aims to obtain a similar result to what is described in this section. The authors use a modified version of the Agent Java PathFinder (AJPF) to track and number all of the states of the agent, but then use this information to generate a Spin or Prism model for the verification process, instead of using the AJPF for the model checking. For the Prism model the probability values are included with a Java class that needs to be specialised with each application. The end result of this process is a Prism model that features a variable that indicates the state number, and where probabilities of transition are defined from the values specified in the new Java class. The case of plan selection when plans share the same triggering event is not contemplated, in which case the agent possibly executes one plan at the time in the order in which they are listed in the agent code. There are a few problems with this approach that the LISA implementation aims to solve: first the process of exploring the model with AJPF to find all the states of the agent is highly computationally expensive and represents a significant bottleneck that makes the model generation quite slow. Furthermore the programmer is required to modify the new Java class for each agent implementation to model the environment.

The process of generating Prism models from the agent code described here is significantly different from the one in [63]. The aim here is to generate a Prism model directly from the agent code without exploring a symbolic model of the agent, which overcomes the drawback of the computational load required to do so. This is made possible by using a completely different approach to the implementation of the Prism model: instead of using a single variable to number all the states of the agent, a Boolean variable for each belief is defined and transition probabilities are taken from the probability distributions defined in the sEnglish code as described in Subsection 3.4.1. In Prism each step will have a different set of values for these variables, which represents in fact a state of the agent, and parallel transitions modify single variables to change the state when appropriate.

The variables of the automatically generated PRISM program can be grouped in the following two main categories:

- *Belief variables.* A variable is defined for each mental note, perception belief and action feedback. Mental notes can be inferred by looking at the plan library of the agent program for internal actions (addition with $+$ or deletion with $-$), and by looking at the '`INITIAL BELIEFS AND GOALS`' section for prepositions that are not perception beliefs. Perception beliefs are taken from the '`PERCEPTION PROCESS`' section, where all percepts are listed. Action feedbacks are found by scanning through all the action '`.sep`' files.

- *Plan index variables.* A variable representing the plan index is defined for each plan of the *Executable Plans* library. The range of the variable is inferred by simply counting the number of actions for each plan. Assigning a variable to each plan index makes the definition of variables for actions unnecessary: actions are only allowed to be used within plans, and there is only one action for each value of the plan index.

PRISM offers the possibility of organising the model in *modules*, which are independent entities within the model that operate on separate variables that other modules have read access to but cannot modify. Transitions defined in separate modules can be executed at the same time in parallel if they are *synchronised.* To do so PRISM offers the possibility of assigning to each transition a *label*: for each time step all the transitions with the same label are executed at once.

The synchronisation feature of PRISM is used in the model described here to divide the reasoning cycle in two sequential steps:

1. *Belief update.* In the first stage, the Current Beliefs set is updated by changing

the value of the variables associated with mental notes, percepts and action feedbacks.

2. *Plan update.* In this stage the plan indices are updated if the state satisfies the required conditions, e.g. the triggering condition is satisfied or action feedbacks of the previous actions have become true.

The synchronisation of the two steps is made possible with a dedicated module that will be named '`scheduler`', showed in Figure 4.2. The use of the variable '`x`' in this way, forces the two transitions to be executed sequentially and consequently all the transitions labelled with '`b`' in the model, in this case *belief updates*, and all transitions labelled with '`t`', in this case *plan updates*, will be sequentially synchronised.

```
module scheduler
x: [0..1] init 0;

[b] x=0 -> (x'=1); //belief updates
[t] x=1 -> (x'=0); //plans updates
endmodule
```

Figure 4.2.: *Example of scheduler module used in PRISM to synchronise the steps of the reasoning cycle.*

In order to modify all the variables that make up the state of the agent at the same time, without implementing complex and numerous transition structures to account for each possibility, these have to be defined in separate modules. In particular there will be a module for each *plan*, which controls the plan index variables, a module for each *action*, where all the variables representing action feedbacks from the same action are controlled, one for each *mental note* and finally one for each *percept*. Additionally, special modules are included to define *reward structures* associated with beliefs or actions throughout the model.

Here follows a brief description of each of these modules and how the implementation was done to reflect the LISA system reasoning and its reasoning cycle as described in Section 3.3.

Plan modules

As mentioned before, to each plan is assigned a variable that keeps track of the state of the plan. Since all plans must be updated at the same time, these variables have to be defined in separate modules so they can be modified during the same transition. However in case two or more plans have the same triggering condition they will be implemented in the same module, as nondeterministic choices for MDP models in PRISM must be defined within the same module.

Figure 4.3 shows pseudo PRISM code for a module of a plan that the translation algorithm generates from the sENGLISH program. Note that the fact that the plan index functions as an identifier for actions makes the definitions of action variables unnecessary. To generate the PRISM code, the translation script analyses each plan in the main reasoning file. At the beginning of the script a structure with each plan is created, associating to each action eventual action feedbacks, which are retrieved from the action files. When generating the PRISM code, the script looks at this structure and creates all the necessary conditions - and negation of conditions - necessary to implement the logic.

All transitions in the plan modules are synchronised with the label '`t`' of the scheduler (see Figure 4.2). The plan index is initialised with a value of 0 (line 2). If the triggering condition is not satisfied the index is kept at 0, otherwise it is advanced to 1 to execute the first action (lines 5-6). The value of n_λ is found by simply counting the number of actions in the plan in the sENGLISH reasoning program. The triggering condition is the first line of a plan and it is expressed here as `<triggering_event> & (<context>)`, where `<context>` is a logic condition on belief variables expressed in the usual manner. When actions are internal, they are considered to be executable within a single reasoning cycle, so there is no need for

```
1  module plan_n
2  plan_n : [0..$n_\lambda$] init 0;

4  //triggering condition
5  [t] plan_n=0 & !(<triggering_condition>) -> (plan_n'=0);
6  [t] plan_n=0 & (<triggering_condition>) -> (plan_n'=1);
7  //internal action
8  [t] plan_n=1 -> (plan_n'=2);
9  //external action
10 [t] plan_n=2 & !(<action_feedbacks>) -> (plan_n'=2);
11 [t] plan_n=2 & (<action_feedbacks>) -> (plan_n'=3);
12 ...
13 //last action
14 [t] plan_n=$n_\lambda$ & !(<action_feedbacks>) -> (plan_n'=$n_\lambda$);
15 [t] plan_n=$n_\lambda$ & (<action_feedbacks>) -> (plan_n'=0);
16 endmodule
```

Figure 4.3.: *Pseudo PRISM code for a plan module of the automatically generated PRISM program. The plan index is initially incremented if the triggering condition applies and then if the related action feedback is true. Note that if the action is not external, the action feedback check is omitted.*

action feedbacks (line 8). When an action is external the plan index is not advanced until at least one of the action feedbacks has a value of 1 (lines 10-11). After the last action has been executed, the index is reset to 0 (line 15). Note that no distinction is made between `runOnce` and `runRepeated` actions: they can both produce actions feedbacks although it is not compulsory for `runRepeated` actions.

Figure 4.4 shows the pseudo PRISM code for two plans that have the same triggering condition. The transitions on lines 7-8 have the same condition, which tells PRISM that it is a nondeterministic choice for the MDP model.

Action modules

In the automatically generated PRISM program, the action modules take care of updating variables for action feedbacks that in turn are used by the plan modules to update the plan indices. Action feedback variables are updated with probability distributions as described in Subsection 3.4.1. For action feedbacks the probability distribution used here is over time, and it is a linear curve that goes from 0 to 1

```
1  module plan_1_2
2  plan_1 : [0..$n_{\lambda_1}$] init 0;
3  plan_2 : [0..$n_{\lambda_2}$] init 0;

5  //triggering condition
6  [t] (plan_1=0 & plan_2=0) !(<triggering_condition>) -> (plan_1'=0 &
       plan_2'=0);
7  [t] (plan_1=0 & plan_2=0) (<triggering_condition>) -> (plan_1'=1);
8  [t] (plan_1=0 & plan_2=0) (<triggering_condition>) -> (plan_2'=1);

10 //actions for plan1
11 [t] plan_1=1 & !(<action_feedbacks>) -> (plan_1'=1);
12 [t] plan_1=1 & (<action_feedbacks>) -> (plan_1'=2);
13 ...
14 [t] plan_1=$n_{\lambda_1}$ & !(<action_feedbacks>) -> (plan_1'=$n_\lambda$);
15 [t] plan_1=$n_{\lambda_1}$ & (<action_feedbacks>) -> (plan_1'=0);
16 //actions for plan2
17 [t] plan_2=1 & !(<action_feedbacks>) -> (plan_2'=1);
18 [t] plan_2=1 & (<action_feedbacks>) -> (plan_2'=2);
19 ...
20 [t] plan_2=$n_{\lambda_2}$ & !(<action_feedbacks>) -> (plan_2'=$n_\lambda$);
21 [t] plan_2=$n_{\lambda_2}$ & (<action_feedbacks>) -> (plan_2'=0);
22 endmodule
```

Figure 4.4.: *Pseudo PRISM code for a plan module for two plans that have the same triggering condition. If the triggering condition applies the system has to make a nondeterministic choice between the transitions of lines 7-8.*

in $2\sigma + 1$ time and it is centred around an average value μ (see Equation 3.7 and Figure 3.5).

Figure 4.5 shows pseudo PRISM code for the implementation of an action module with a single action feedback. The translation script scans the sENGLISH project folder for action files and creates a structure with every action, its associated action feedback(s) and the probability distribution values that describe the activation of the action feedbacks. When creating the PRISM module for the action, the script also scans the structure holding the plans list so to generate the necessary conditions of activation for the action feedbacks.

```
1  module <action_name>
2  af: [0..($\mu+\sigma+1$)] init 0;

4  //activation
5  [b] !(<plan_indices>) & af<=1 -> (af'=0);
6  [b] (<plan_indices>) & af<=1 -> (af'=2);
7  //dead zone
8  [b] af>1 & af<=($\mu-\sigma$) -> (af'=af+1);
9  //transition
10 [b] af>($\mu-\sigma$) -> $\frac{\text{af}-(\mu-\sigma)}{2\sigma+1}$ : (af'=1) + (1-$\frac{\text{af}-(\mu-\sigma)}{2\sigma+1}$) : (af'=af+1);
11 endmodule
```

Figure 4.5.: *Pseudo PRISM code of an action module for an action with a single feedback (*`af`*). The implementation of the probability distribution is activated when the plan index reaches a value associated with the action.*

The action feedback variable itself is used as a counter. It is initialised with a value of 0, when one of the plan indices that the action is associated with becomes of the right value, the variable is updated to 2 (line 6). The progression is shifted positively of 1 because the value 1 itself is used to detect when the action feedback is true in the plan modules. The value is then increased at each reasoning cycle until it reaches the lower end of the range with a value of $\mu - \sigma$ (line 8). Once the variable reaches the desired range, the transition is programmed to have a probability value according to Equation 3.7 - evaluated at $t-1$ - of activating the action feedback, with of course the complement probability of not activating included to avoid deadlocks (line 10).

The case when an action is programmed to have more than one action feedback is implemented in a similar way. In this case the first action feedback is used as a counter and the only difference is in line 10 where the probability value is split according to the weighting factors specified in the '`.sep`' file. For example, in case there are two action feedbacks `af_1` and `af_2` the PRISM model becomes:

$$
\begin{aligned}
\texttt{[b] af\_1>}(\mu-\sigma)\ \texttt{->}\ & p_1 \cdot \frac{\texttt{af\_1} - (\mu-\sigma)}{2\sigma+1} \ \texttt{: (af\_1'=1 \& af\_2=0)} \\
& \texttt{+ } p_2 \cdot \frac{\texttt{af\_2} - (\mu-\sigma)}{2\sigma+1} \ \texttt{: (af\_1'=0 \& af\_2=1)} \\
& \texttt{+ (1-}\frac{\texttt{af\_1} - (\mu-\sigma)}{2\sigma+1}\texttt{)} \ \texttt{: (af\_1'=af\_1+1);}
\end{aligned}
$$

where p_1 is the weighting factor for `af_1` and p_2 is the weighting factor for `af_2` with $p_1 + p_2 = 1$. As usual, μ and σ represent average number of reasoning cycle and variance, common for both action feedbacks.

Mental note modules

The mental notes of the LISA system are updated in the first step of the reasoning cycle, but before they are stored into the Current Beliefs set, the logic based implication rules are applied to them. Actions can modify mental notes in two ways: addition or deletion. In this framework that means they can make the variable associated to mental notes either equal to 1 or 0.

Although beliefs are gathered and logic rules are applied sequentially, the *mental notes* updates are still modelled within a single synchronised step (`b`), so to avoid the creation of a state space that is bigger than required. The logic behind this implementation is shown in the truth tables of Table 4.1. In this implementation rules are always given priority over actions, as they are applied to the beliefs after the first step of sensing, and actions/rules that change the value of the mental note

Table 4.1.: *Truth tables for the update of the mental notes.* M *stands for mental note,* A *for action and* R *for rule. The '+' superscript indicates change to* 1 *of the mental note while '−' indicates change to* 0*.*

$M[t-1]$	A^+	R^+	R^-	$M[t]$	$M[t-1]$	A^-	R^+	R^-	$M[t]$
0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	1	0	1	1	0	1	0	1
0	0	1	1	1	1	0	1	1	0
0	1	0	0	1	1	1	0	0	0
0	1	0	1	0	1	1	0	1	0
0	1	1	0	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	0

are given priority over action/rules that do not. This results in the logic condition $(A^+ \wedge \overline{R^-}) \vee R^+$ when the mental note variable is currently 0, and $(A^- \wedge \overline{R^+}) \vee R^-$ when the mental note variable is currently 1. Figure 4.6 shows how this is implemented with pseudo PRISM code.

```
module mn
mn: [0..1] init 0;

[b] mn=0 & !((<plan+> & !<rule->) | <rule+>) -> (mn'=0);
[b] mn=0 & ((<plan+> & !<rule->) | <rule+>) -> (mn'=1);
[b] mn=1 & !((<plan-> & !<rule+>) | <rule->) -> (mn'=1);
[b] mn=1 & ((<plan-> & !<rule+>) | <rule->) -> (mn'=1);
endmodule
```

Figure 4.6.: *Pseudo PRISM code of a module for a mental note* `mn`*. The logic is implemented according to Table 4.1.*

Plan conditions are simply found by scanning through the '`EXECUTABLE PLANS`' section in the sENGLISH main reasoning file for actions that start with a '+' or '–' symbol. The condition is then implemented as the name of the variable of the corresponding plans equal to the index of the action that activates or deactivates the mental note. Similarly for the rules, the translation script scans through the Reasoning section of the main sENGLISH file and looks for rules that change mental notes.

Percept modules

Although action feedbacks and percept modules are modelled in a similar way, the PRISM model needs to be constructed differently. In the case of the action feedbacks the variable declared to represent them was used as a counter for the number of reasoning cycles elapsed since the action is called. This is possible because the value of the action feedback itself is only relevant when it is activated and it is deactivated straight away after a single reasoning cycle. When modelling the perception beliefs however a different approach must be taken as the value of the percepts is always relevant throughout the agent program, for plan contexts and rules as well as triggering events. Furthermore, different probability distributions can be defined for activation and deactivation of the percept, therefore using the variable itself as a counter is not easily implementable. This means that a new variable must be declared to act as a counter.

The translation script gathers all the necessary information from the main reasoning file. By scanning the '`PERCEPTION PROCESS`' section of the main reasoning file, a structure with all the percepts is created, completed with activation and deactivation probability distribution values and the conditionality list.

For each percept the probability distribution is copied at interval equally spaced according to the mean value μ (see Equation 3.9). The distribution chosen for the percepts, depicted in Figure 3.7, is composed by two linear segment, increasing from 0 to the specified probability value p between $\mu - \sigma$ and σ, where σ is the specified variance, and decreasing to 0 between μ and $\mu + \sigma$.

Figure 4.7 shows pseudo PRISM code for the implementation of percept modules. Conditionality on other percepts is always checked and the percept variable and the counter are reset in case the condition expressed with `<cond>` does not apply (line

6). When the percept has a value of 0, the *activation* probability distribution is applied. First, when the conditionality applies the counter is increased through the dead zone, that is between 1 and $\mu - \sigma$ (line 8), in which case the probability of activation is equal to 0. Second, the probability defined in Equation 3.9 is applied: line 9 for the increasing part and line 10 for the decreasing part of the distribution. The *deactivation* is handled in the same way (lines 12-14). Finally the counter is reset when it goes out of range (line 16).

```
1  module sp
2  sp: [0..1];
3  c_sp: [0..($\mu+\sigma$)];

5  //conditional probabilities not met
6  [b] !<cond> -> (sp'=0) & (c_sp'=1);
7  //activation
8  [b] sp=0 & c_sp<($\mu-\sigma$) & <cond> -> (c_sp'=c_sp+1);
9  [b] sp=0 & c_sp>=($\mu-\sigma$) & c_sp<=$\mu$ & <cond> -> $p_s \cdot \left(1+\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$ : (sp
       '=1) & (c_sp'=0) + (1 - $p_s \cdot \left(1+\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$) : (c_sp'=c_sp+1);
10 [b] sp=0 & c_sp>$\mu$ & c_sp<=($\mu+\sigma$) & <cond> -> $p_s \cdot \left(1-\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$ : (sp
       '=1) & (c_sp'=0) + (1 - $p_s \cdot \left(1-\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$) : (c_sp'=c_sp+1);
11 //deactivation
12 [b] sp=1 & c_sp<($\mu-\sigma$) & <cond> -> (c_sp'=c_sp+1);
13 [b] sp=1 & c_sp>=($\mu-\sigma$) & c_sp<=$\mu$ & <cond> -> $p_s \cdot \left(1+\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$ : (sp
       '=0) & (c_sp'=0) + (1 - $p_s \cdot \left(1+\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$) : (c_sp'=c_sp+1);
14 [b] sp=1 & c_sp>$\mu$ & c_sp<=($\mu+\sigma$) & <cond> -> $p_s \cdot \left(1-\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$ : (sp
       '=0) & (c_sp'=0) + (1 - $p_s \cdot \left(1-\frac{\text{c\_sp}-\mu}{\sigma+1}\right)$) : (c_sp'=c_sp+1);
15 //counter overflow
16 [b] c_sp>($\mu+\sigma$) -> (c_sp'=0);
17 endmodule
```

Figure 4.7.: *Pseudo PRISM code of a module for a sensory percept* sp. *Activation and deactivation of the percept are implemented separately.*

Assuming the LISA program is correctly defined, all the necessary information for the definition of the percepts modules is found in the main sENGLISH reasoning file under the '`PERCEPTION PROCESS`' section.

Rewards

Reward structures are defined in the LISA program by listing the name of the reward and the value associated with it. This information is easily converted to standard definition of reward structure in PRISM by the translation script.

Figure 4.8 shows pseudo PRISM code that illustrates how a reward is implemented in the automatically generated PRISM program. In this implementation, `<condition_j>` can be on plan indices or a beliefs. The value expressed with `<value_j>` is a number that will be added to the reward when the condition applies.

```
rewards "<reward_name>"
<condition_1> : <value_1>;
<condition_2> : <value_2>;
...
endrewards
```

Figure 4.8.: *Pseudo PRISM code for the definition of reward structures.*

The same structure is repeated for each different reward specified in the reasoning file.

4.4. Design-time verification

Section 4.3 described how a PRISM model of the system is automatically generated from the agent code for the purpose of verification. Once the model is defined, the verification process is fairly straightforward. PRISM offers a great variety of tools for verification, for example an integrated simulation environment where the user can generate specific path or simulate them automatically. The specifications for verification are defined with an extended version of PCTL as described in Subection 2.2.2.

The approach used here to implement the agent reasoning as a discrete model is to use separate variables for each belief and each plan index. This allows to explicitly define arbitrarily complex properties that can touch any aspect of the reasoning process. The model checker will then generate counterexamples [52, 98], traces in the model that do not satisfy the PCTL specification defined by the user. This information can be used to improve the agent program itself as an iterative process.

For example, assume that an agent is implemented to have two actions that are opposite to each other and that should never be executed at the same time, such as '`go left`' and '`go right`'. Even though the designer can be careful not to make these two actions execute simultaneously, a model checker is the only way to *guarantee* that this situation will never happen during any execution, or at least that the probability of it to happen is contained in a limited range. For example assuming that the agent is programmed to have $\pi_2(1)$='`go left`' and $\pi_4(2)$='`go right`', the property:

$$\mathtt{P}_{\mathtt{max}=?}\,[\mathtt{F}\;\;(\mathtt{plan\_2} = \mathtt{1}\;\&\;\mathtt{plan\_4} = \mathtt{2})]$$

will ask the model checker to generate "the maximum probability of '`go left`' and '`go right`' to be executed at the same time at some point in the future".

The result of an iterative design-time verification process is an improved agent code, which is corrected against the properties used during the verification, and a discrete model of the code. In Section 4.5 a method is proposed to use this model to improve the decision-making capabilities of the agent reasoning.

4.5. Run-time verification

The *internal model* of a system is an internal mechanism for representing both the system and the environment, that is then used by the system itself to improve its own performances. In [110] it is argued that an internal model allows a system to look ahead to future consequences of actions, without committing itself to said actions. This is not unlike human decision-making: in most cases we first ponder a number of options by making predictions on their possible outcomes, and then we commit to the one we consider the most suitable according to our current beliefs. A few example of applications of this concept can be found in [31, 65, 156].

The discrete model of the LISA system described in this chapter falls under the definition of internal model as it includes the behaviour of the agent reasoning as well as its interface with the world. If the verification process is performed at run-time, the model can be used as a means of evaluation, in a probabilistic fashion, of the outcome of possible plan choices. In this section two different methods are proposed for using a run-time verification process for this purpose.

In many cases, most of the computational power required to verify such models is usually spent by the model checker on building the model itself, which does not influence the verification time. In other words, once the model is built, the user can run different verification queries without having to rebuild the model. Modern probabilistic model checkers such as PRISM allow to verify fairly large models in a matter of seconds, making the run-time verification process a feasible technique to use to apply the concept of internal model for improving the decision-making capabilities of the agent.

The first method is to implement the run-time verification process as a *skill* of the agent, e.g. as a module of the full system, as illustrated in Figure 4.9. The DTMC or MDP model is verified against a set of predefined queries. In particular, in PRISM, it is possible to check a query by selecting a starting state with the use of *filters* [205]. The run-time verification is then used to generate a set of results that will be interpreted by a '*generate beliefs*' function that will activate or deactivate certain beliefs in the agent Current Beliefs set.

This is in principle a skill of the agent that would be initialised at the beginning

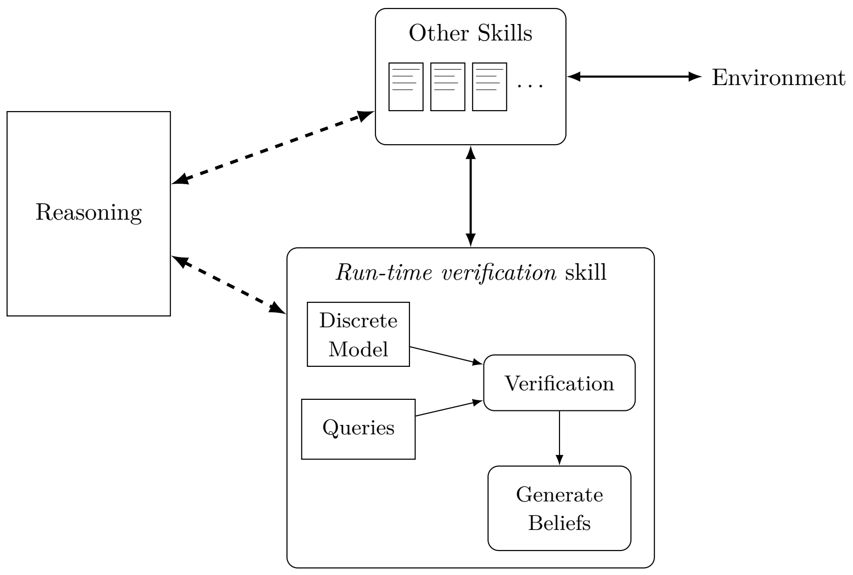


Figure 4.9.: *Implementation of the run-time verification process as a skill of the agent. The verification process is used to activate or deactivate beliefs of the agent reasoning based on a set of pre-defined queries that are run against the discrete model.*

of the execution and that would run continuously, giving the agent reasoning a quantitative estimate on consequences of future actions therefore a deeper knowledge on the state of the world, possibly improving its decision-making capabilities.

An example of verification query for a mobile robot could be:

$$\texttt{R}\{''\texttt{fuel}''\} \geq \texttt{100}\ [\texttt{F}\ \texttt{mission\_complete} = \texttt{1}]$$

which returns 1 when "the expected reward value for '`fuel`' cumulated up until '`mission_complete`' becomes 1 at some point in the future, is greater than or equal to 100".

The LISA implementation of the BDI-based reasoning cycle uses a multi-threaded workflow to avoid the need of defining a set of functions that are external to the agent code. However one of these external functions, the Plan Selection function, still plays a role in the LISA reasoning operation. When the programmer chooses to implement the plan library with plans that have non-unique triggering conditions, the Plan Selection function is required to select one of the plans that are triggered by the same triggering condition. In Jason, the Plan Selection function and the aforementioned additional functions have to be developed in JAVA by overriding default classes of the underlying structure of the agent. The second method proposed here for using run-time verification in this framework consists of implementing a *Plan Selection function* that makes use of model checking to assess probability of success of a set of options based on user-defined specifications, and selects the most suitable plan. A clear advantage of this approach is that, since the probabilistic model is generated automatically, the user does not need to implement a specialised Plan Selection function for each agent, therefore making the development process more focused on the logic reasoning of the agent.

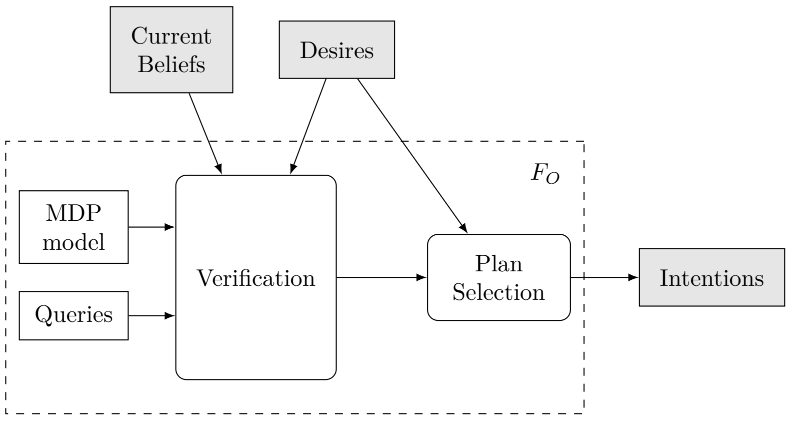


Figure 4.10.: *The* Plan selection *function implemented as a run-time verification process. The verification process is used to generate probabilities of success for the desired plans, which help the agent reasoning decide amongst plans that are triggered by the same conditions.*

Figure 4.10 illustrates the structure of the Plan Selection function implemented as a run-time verification process. The structure sits within the reasoning cycle depicted in Figure 3.2 however in the final operation of the agent this function is still external to the agent program. Ideally the run-time verification will be fast enough to be executed at least once for each reasoning cycle. However an even faster execution time might be needed when more than one event triggers multiple plans. A possible practical solution is that if the plan selection requires more than one reasoning cycle, the agent suspends the decision until results of the run-time verification are available.

PRISM allows to run verification starting from a state that is not the initial state with the use of so called *filters*. With the MDP model that was generated at design-time and the set of Current Beliefs, a set of queries that can be automatically generated or preprogrammed by the developer is used to perform the run-time verification and generate probability values associated with each desired plan. These probability values are then used by a simple Plan Selection routine that selects one of the plans based on some predefined criteria, for example minimising the probability of failure. Once the most suitable plan for the current situation has been selected, it is passed on to the Intentions set for execution.

Even though the second method proposed here is limited to be used only when there are multiple plans that share the triggering condition, i.e. the Plan Selection function is needed, the two methods are not in principle mutually exclusive. With the second method the agent could still be programmed to have a run-time verification skill, assuming that the computational power of the machine the agent is implemented on allows for such a load. In both applications the run-time verification gives the agent additional knowledge about the world in the form of probabilistic estimates so that the agent can use this information to improve its own effectiveness.

4.6. Conclusions

The process of abstraction of the agent reasoning of the Limited Instruction Set Agent (LISA) system to finite state machine for the purpose of design-time and run-time verification was described in this chapter with detailed explanations of how the agent program can be automatically translated to a model for the probabilistic model checker PRISM.

In order to perform verification of the agent reasoning a complete model of the system is needed, one that include both the logic reasoning of the agent and its interface with the environment. Two discrete probabilistic models are chosen for the purpose: Discrete-Time Markov Chain (DTMC) and Markov Decision Process (MDP). Discrete time models can be considered ideal as the state of the agent reasoning is only relevant at the end of each reasoning cycle, where beliefs, events and plan indices are already processed and completely updated. Probabilistic models are necessary to abstract the BDI agent based system and its environment as input variables change in a probabilistic fashion. LISA reasoning is proven to be abstractable as a DTMC when all plans in the Plan Library feature unique triggering conditions. The LISA system was proven to be modellable as a MDP for any well defined implementation of the agent code.

The process of translating the agent program to a probabilistic model in the input language of the model checker PRISM is made possible with the additional information included in the agent program as described in Section 3.4. This gives several advantages over previous attempts at the verification of BDI agent reasoning in terms of performances and flexibility. In particular this approach makes possible to avoid the execution of a simulation of the agent for the purpose of constructing the state space and allows the programmer to focus on the development of the agent logic without having to implement external libraries to include the probabilistic behaviour of the environment.

Once a model of the system is constructed from the agent code, it can be used to perform verification by model checking at design-time. Verifying specifically designed queries allows to pinpoint design flaws in the agent program by analysing the counterexamples generated by the model checker. This in turn allows to improve the agent program by correcting design flaws in an iterative way.

A model constructed in this way can also be used for run-time verification. The verification process can be used to make probabilistic estimates of future outcome of actions. This can be done in two ways in this framework: the first method is to implement a skill of the agent that includes the verification process and generates beliefs for the agent based on the probabilistic information. The second method is to implement the Plan Selection function as a run-time verification process. Run-time verification initialises the model according to the Current Beliefs and generate plan success probabilities that are then used to select a plan that optimises performances or minimises failure rates.

Chapter 5.

Implementation and simulation

The flexibility and accessibility of agent-based systems makes them highly suitable for practical applications. The logical nature and the clear schematics of natural language agent programs of the LISA system, allows implementations in a variety of software environments for simulation purposes as well as real-world applications. Furthermore, the modularity of the agent architecture makes the integration of the agent reasoning with its skills an accessible process, as skills are not bound to be implemented in any particular language.

5.1. Introduction

The BDI agent-based system named LISA, described in this thesis, is a tool that can prove useful in a variety of real-world applications. This chapter describes several possible approaches to the implementation and simulation of the LISA system.

Generally speaking, the agent reasoning of a BDI-based agent is an iterative function that takes inputs at each iteration, updates its internal state and deliberates on commands to be executed based on said inputs. In the case of the LISA system, the agent reasoning is mainly a logic program which does not rely on any particular feature from any language, and it can therefore potentially be implemented using a large variety of programming languages as long as they support basic Boolean logic operations. However given the numerous implementations variety of agent frameworks and applications available to date, implementation of a stand-alone application that runs the agent logic of LISA, with interfaces to a simulation environment, was not considered to be within the scope of this work.

The most used BDI-based agent implementations are made using JAVA environments, see for example Jason [37] or Gwendolen [60]. By compiling sENGLISH code using the Cognitive Agent Toolbox (CAT) [214] it is possible to generate Jason compatible code that can be executed in a simulation environment as will be explained in Subsection 5.2.1.

Given the schematic representation of LISA reasoning with the sENGLISH language, it is also possible to automatically generate a MATLAB function that implements the agent reasoning and run it continuously in a SIMULINK model. Similarly all the skills can be implemented as MATLAB functions and included in the simulation model in SIMULINK. A description of this approach to the implementation of the LISA system is given in Subsection 5.2.2.

Another possible approach to the implementation of the LISA system is to implement it within popular robotic software packages such as Robot Operating System (ROS) [180, 206] or Mission Oriented Operating Suite (MOOS)[165, 203] (in particular with the addition of Interval Programming (IvP) [23, 204]). Although both ROS and MOOS-IvP feature the possibility of running simulation environments for system testing, they are mostly used in practical applications, and implementing the LISA reasoning and its skill as nodes of these software structures can be a great tool for bringing the agent architecture to real-world scenarios. The possibility of implementation of the LISA system in ROS or MOOS-IvP is described in Subsection 5.2.3.

Simulation of any system, especially the ones that are going to work in safety critical environment, is a great tool to test its capabilities and operation in a set of controlled situations. For physical systems, such as autonomous robots, the simulation environment must include a dynamical model of the machine itself together with a model of the environment it will be operating in. A visual reference in simulation can be a valuable tool that gives the developer an immediate feedback on how the machine is doing in the simulated environment. Graphical models are possible in both the SIMULINK related approaches and the robotics software approaches, ROS and MOOS-IvP.

5.2. Implementation approaches

Most agent-based systems are composed of two main parts: the agent reasoning and the agent skills. Given their intrinsically different nature the two parts can be - and often are - developed using different languages. In the case of the LISA system the agent reasoning is an iterative process that makes logic deliberations on symbolic data coming from its skills at each iteration (reasoning cycle). The program that describes the agent reasoning of the LISA system, the agent program, is developed with the NLP language sENGLISH.

Given the simplicity and schematic description of the agent reasoning, there are clearly many ways to compile the agent program into a function to be run on specific hardware, and in principle there is no particular advantage in using one method or another. Since skills can also potentially be implemented in any language, building an interface between the agent reasoning and its skills is particularly important.

This section explores a set of solutions for developing the agent reasoning and its skills in unified environments that lend themselves well for use with either simulations or real-world applications.

5.2.1. Implementation with the Cognitive Agent Toolbox

The sEnglish language and the Machine Ontology Language (MOL) are part of a software suite called Cognitive Agent Toolbox (CAT) developed by SysBrain Ltd. The first natural way to implement the LISA system is by using the CAT, which features a two part system development: an Eclipse Application Programming Interface (API) and a Matlab toolbox called Agent Executive Toolbox (AET). The Eclipse API, called *sEnglish Publisher*, allows to develop agent programs in sEnglish and to compile them in Jason+, a modified version of Jason. The code for the skills of the agent, which can be included in the action files of the sEnglish project, can automatically be compiled into Matlab function files. The process is outlined in Figure 5.1.

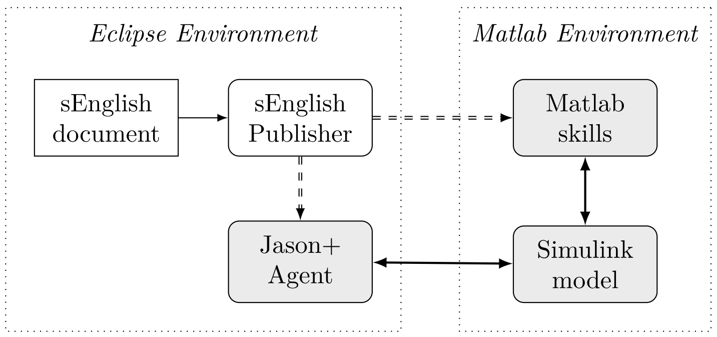


Figure 5.1.: *Illustration of the Cognitive Agent Toolbox. Simulation is carried out by running the grey shaded blocks, linked together with the Agent Executive Toolbox.*

The simulation of the full system is made possible with the AET. The Jason+ agent runs as a Java application and actions are translated as function calls for the AET to execute Matlab functions which implement skills. In the sEnglish program, action sentences can include variable formats specified in the ontology file as MOL. This information is used by the AET to coordinate and run the skills with the appropriate inputs/outputs. The actions can also be programmed as `runOnce` or `runRepeated` as explained in Section 3.3.1.

In Section 4.5 two approaches for using a run-time verification process to enhance the decision-making capabilities of the agent were proposed: the first was to implement the verification process as a skill and the second was to implement a Plan Selection function that uses the verification process to select plans based on probabilistic results generated by it. One of the great features offered by the PRISM software is that models expressed in PRISM input language can be exported in the form of matrices to be used with other tools. In particular the user can export the *set of reachable states*, the *transition matrix*, the *reward structures* and so on (see the manual section in [205] for further details). Among similar software, MATLAB is the framework that offers possibly the widest variety of tools to work with large matrices in a very simple and accessible way. Since all the skills are implemented as MATLAB functions, this makes the implementation of the verification process as a skill of the agent fairly straightforward once the user takes care of exporting model matrices from the automatically generated PRISM model. As for the inclusion of the verification process as part of the Plan Selection function, the programmer would still be forced to modify the underlying JAVA structure of Jason, which is still possible but it would undermine the principle of a unified workflow introduced with the LISA system.

This framework lends itself particularly well for simulation purposes. The fact that most of the architecture resides in MATLAB/SIMULINK provides the programmer with the possibility of developing arbitrarily complex dynamical models that represent the physical behaviour of the robot in its environment, in the intuitive and flexible MATLAB/SIMULINK environment. The model can in turn be used to generate adequate sensorial input for the skills to convert into symbolic information that the agent reasoning can use to make deliberations and activate/deactivate other skills when needed.

An obvious drawback to this approach is that the agent reasoning is still implemented as a Jason agent. This means that some of the features of LISA cannot be used in the final product. For example the multi-threaded workflow described in Section 3.3.1 will not be translated to Jason. However assuming that the user does not make use of some of Jason's features such as personalising external functions (F_M, F_E or F_I, see Section 3.3.2 for details), the model described in Chapter 4 is still valid and the automatic verification techniques can still be used.

5.2.2. Direct implementation with Matlab/Simulink

The implementation of the LISA with the CAT is possible by implementing all the skills as Matlab functions and running the agent logic as an external stand-alone application. A possible valid alternative is to also implement the logic of the agent as a Matlab function, so to keep everything within the same environment and prevent any interface problem.

In Section 4.3 the agent code was used to generate Prism models for verification. This is always possible with the agent program of a LISA as it contains all the necessary information to generate a complete model. In a similar fashion the sEnglish program can be used to generate a Matlab function that reproduces the agent logic and that can be run within a Simulink diagram.

Figure 5.2 shows a snippet of the Simulink diagram for an implementation of the LISA system within a simulation environment. The agent logic is running in the *Matlab function* block. At each time step of execution Simulink runs the functions that read the input signals coming from the skills and updates the output signals. The actions are Boolean variables in the logic program that activate blocks further down the chain. In particular `runOnce` actions activate *triggered* subsystems that detect the rising edge of the signal and execute a Matlab function or any Simulink diagram once, `runRepeated` actions activate *enabled* subsystems which execute the function continuously as long as the input remains active. In most cases the agent reasoning will not be designed to run as quickly as the skills, the `Rate transition`

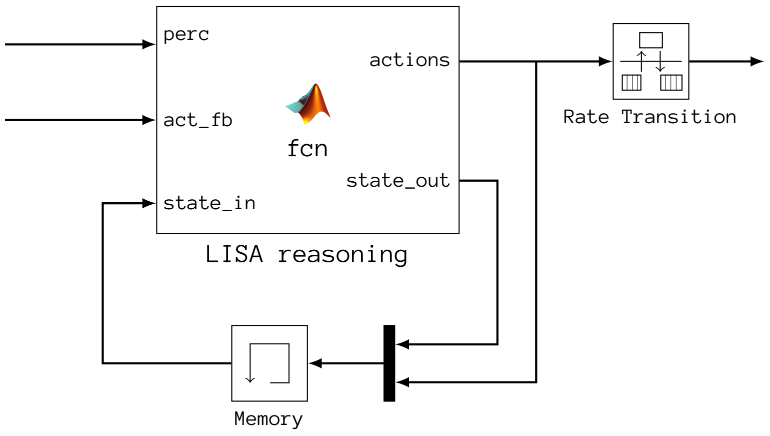


Figure 5.2.: *Partial Simulink diagram of the implementation of the LISA reasoning. The reasoning cycle is implemented as a* MATLAB *function, the state of the agent reasoning is stored thanks to the* `Memory` *block. Skills can run at different sampling times thanks to the* `Rate Transition` *block.*

block simply interfaces parts of the model that work at different sampling times. In order to remember the state of the agent reasoning, the latter is passed through a `Memory` block which delays its input signal by one sampling time step. The action variables are saved as well because `runRepeated` action variables need to remain true to keep the associated actions running.

A clear advantage of this approach to the implementation of the LISA system is that the full architecture is part of a single SIMULINK diagram. This makes the LISA reasoning fully compatible with the rest of the system with a seamless communication with its skills. For this reason the simulation of a full system that includes a dynamical model of the physical system can be performed without steering the attention of the programmer towards interfaces and compatibility issues.

MATLAB/SIMULINK also offers the option to compile the model into an executable C/C++ program. By using this tool the LISA system can be easily ported to for example embedded systems for use in small robotics applications.

Similarly to the approach presented in the previous Subsection 5.2.1, a run-time verification process can be implemented as a skill of the agent by using state and transition matrices exported from the automatically generated Prism model. This is possibly easier in this case as the skills generate outputs that are passed directly to the agent reasoning without external interfaces. Additionally in this case it is also possible to implement a personalised Plan Selection function that includes run-time verification processes within the agent program itself.

The possible drawback with this approach is that as of the time of writing, Simulink does not offer a way to multitask within the same model, or in other words different blocks within the same model cannot execute independently. For example if a block implementing a function takes more than a time step to execute, other blocks within the model will not be able to keep running but they will wait for the function to conclude. This means that although the agent reasoning and its skills are running at different sample times, this is still a sequential execution so for each execution of the agent reasoning there will be a fixed number of steps for the skills to execute and the agent reasoning will hold until they are finished.

5.2.3. Implementation with other software architectures

Another possible approach to the implementation of the LISA system is to develop it as an application of existing frameworks commonly used in robotics. For the purpose of this application ROS and MOOS-IvP are considered, being arguably the most widely used in the robotics community to date.

ROS

ROS is a lightweight architecture that resembles that of a conventional operating system, and applications are developed as nodes of the structure and can communicate and exchange data with other nodes, in a peer-to-peer fashion. ROS is free and open source, and it is supported by a very large and vibrant community. In the robotics community, developers come from widely different backgrounds and have preferences for some programming languages over others. For this reason ROS supports applications written in several languages including C++, Python and Octave. The agent reasoning of the LISA system can be implemented as a ROS application as well as any skill that may be required, if compiled to one of the supported languages. For example with the approach described in Section 5.2.2 there is the possibility of compiling the agent as a C++ application that can then be used as a node of the ROS architecture.

The CAT described in Subsection 5.2.1 also offers two ways of implementing the LISA system, with the reasoning described in sEnglish, as a ROS application. One is to compile the agent reasoning as a Jason+ application and interface it with the skills that are compiled as regular ROS nodes. The second, which is under development at SysBrain Ltd, is to compile the agent reasoning as a ROS application.

The ROS architecture also supports Gazebo [89], an open-source robot simulator that offers a variety of physics engines, 3D graphics, sensor noise generations, supported by an active community of developers. This tool can be used as part of the development process of LISA-based systems to simulate the agent as part of a realistic simulated environment.

MOOS-IvP

Another suitable framework for the implementation and simulation of a LISA system is with MOOS-IvP [23, 165, 203, 204]. MOOS-IvP is one of the most used architectures in the field of marine vehicle autonomy, some examples can be found in [24–26, 97]. MOOS is an inter-process communications middleware software that is structured in a star-like fashion. The structure of the architecture features a central node called the MOOS Database (MoosDB) and a set of applications that communicate with each other through the MoosDB in a publish/subscribe manner. IvP is a MOOS application that implements a behaviour-based architecture over a set of control variables, for example “direction” and “speed”. Behaviours are internal modules of the IvP application that reproduce a particular action over a set of control variables $c_1, c_2, \ldots, c_n$, and generate at every cycle a piecewise linear function called “IvP function” $f(c_1, c_2, \ldots, c_n)$ that maps points of the decision space to values that reflect the degree to which that control array supports the action. Once these functions are produced, a multi-objective optimization problem is solved by another internal module called the IvP-solver:

$$\begin{aligned} \arg\max_{c_1,\ldots,c_n} \quad & w_1 f_1(c_1,\ldots,c_n) + \cdots + w_k f_k(c_1,\ldots,c_n) \\ \text{s.t.} \quad & f_i \text{ is an IvP piecewise defined function} \\ & w_i \in \mathbb{R}_{\geq 0} \end{aligned} \tag{5.1}$$

where $w_1, w_2, \ldots, w_n$ are called *priority weightings*. In MOOS-IvP the priority weightings are influenced by two main factors:

1. With every behaviour is associated a set of binary flags that give control over the activation time of the behaviour itself. These flags can be conditionally modified by the behaviour itself, which means that the behaviour has partial control over its own state, and can also be associated with external variables that can be modified by other nodes.

2. A hierarchical mode system is defined within IvP that allows to organise the behaviour activation according to declared mission modes. Modes and sub-modes can be declared in line with the designer’s own concept of mission evolution, and conditional statements can be implemented so to switch between modes. Modes can also be associated with external variables that can be modified by other nodes.

Figure 5.3 illustrates a possible architecture of the LISA system in the MOOS-IvP framework. The agent reasoning and each set of skills are implemented as

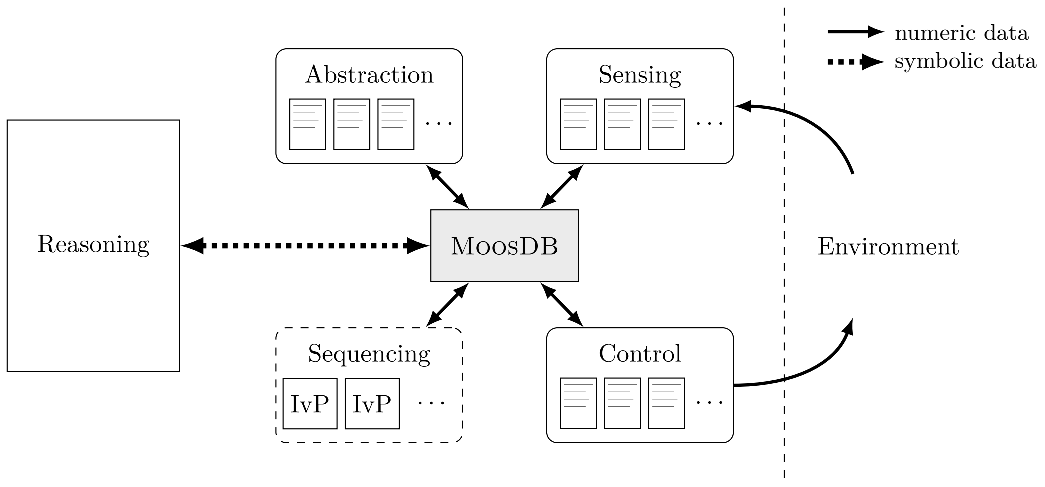


Figure 5.3.: *Implementation of the LISA architecture with MOOS-IvP. The agent reasoning and the skills communicate with each other through the MOOS database.*

MOOS applications. In MOOS communication between nodes happens through the central MoosDB. IvP modules are used as Sequencing skills and they operate with an intermediate loop with data abstracted and filtered from physical sensors. IvP modules operates on a given set of control variables, which are then translated into physical actions by the Control skills.

In this framework the agent reasoning can be implemented to direct and coordinate the decision-making of IvP in two ways. The first is to implement some actions of the agent reasoning as external function that activate or deactivate full IvP modules, for example in situations when the system does not need an intelligent planner for certain variables. Another way is to implement actions of the agent reasoning as external functions that activate, deactivate or modify the weighting factors of behaviours within IvP modules.

5.3. Conclusions

In this chapter a number of ways to implement the Limited Instruction Set Agent (LISA) architecture were described. The schematic representation of the agent reasoning provided by the sEnglish natural language, makes the implementation of the LISA reasoning possible in a variety of different software environments. This allows to introduce the advantages of agent reasoning in many different simulations and real-world applications.

The agent reasoning of the LISA system can be implemented in any language that supports Boolean logic, and thanks to the schematic representation of sEnglish, and the clear structure structure of the reasoning cycle, compilation and translation tools can easily be implemented to adapt the LISA reasoning to potentially any software environment.

The main tool for implementation of the LISA system comes with the sEnglish package and it is called the Cognitive Agent Toolbox (CAT). The CAT provides an API called the sEnglish Publisher that includes compilers to Jason+, a modified version of Jason, and to Matlab. The CAT also includes a tool that allows the compiled Jason+ agent to communicate with a Simulink model which can be integrated with dynamical models for simulation purposes. An alternative to this framework is to implement the agent reasoning as a Matlab function, instead of a standalone application, and still use the excellent simulation tools offered by Simulink. In this case as well a translation tool from the sEnglish program to the Matlab implementation is a reasonably straightforward process. The LISA system also lends itself well to be implemented with popular robotic software such as ROS and MOOS-IvP. The interface with ROS is officially supported with the CAT, and it works in one of two ways: as a standalone application interfaced with skills implemented as nodes of ROS, and as a C application as a ROS node. The agent reasoning and its skills can be similarly integrated in MOOS-IvP, with the reasoning implemented as a MOOS application, and IvP modules used as Sequencing skills to direct and coordinate lower level control algorithms with optimised outputs.

Chapter 6.

A case study

This chapter describes a case study for the application of the LISA system investigated in this thesis, with an implementation of agent reasoning and a simulation environment. The scenario considered is a mine detection and disposal mission for an Autonomous Surface Vehicle (ASV). The autonomous marine vessel explores a predefined area, making sure the area is fully covered, and tags spots representing potential threats.

6.1. Introduction

Consider an Autonomous Surface Vehicle (ASV) [198] with the purpose of mine detection and disposal. The mission, schematically depicted in Figure 6.1, is to explore an area at sea that contains a number of mines and take actions for disposal of potential threats. The ASV communicates with a control centre that could be either another boat or an ashore facility, where humans can monitor the outcome of the mission with the ASV giving information and motivations on the decisions taken. It is assumed here that the ASV starts and complete its mission from the control centre.

The ASV is built so to minimise the electromagnetic signature and avoiding setting off mines, that are usually designed to damage much larger vessels by following their

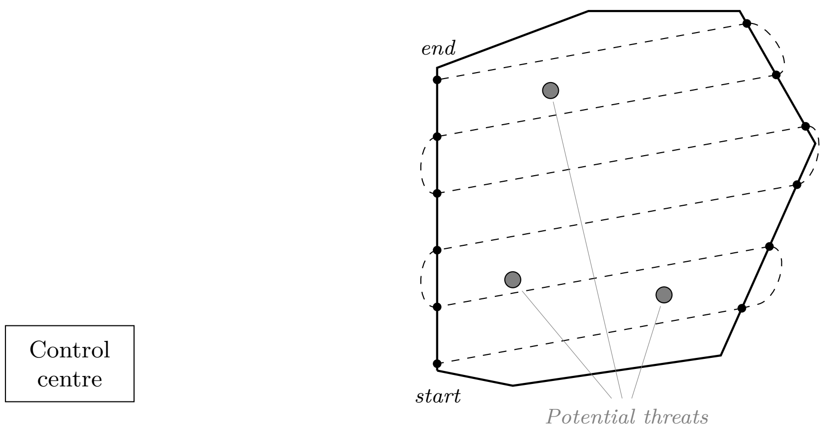


Figure 6.1.: *Illustration of a mission for mine detection and disposal.*

electromagnetic influence. The main mission of the ASV is to survey and completely cover the area, as illustrated in Figure 6.1.

The ASV is equipped with sensing equipment such as sonars and cameras that allow the detection of unidentified objects in the area of interest. All the data that is collected during the mission is continuously sent back to the control centre. Once the ASV detects and tags objects that might be dangerous, potentially mines, human operators in the control centre will analyse pictures and data in order to decide whether or not the object need further investigation or intervention.

The sensing equipment gives the ASV a cone shaped visibility range. Given that the shape and size of the visibility range is known in advance, the algorithm that generates the lawn mower surveying path will be able to distantiate the individual tracks so to completely cover the area of interest. In this example a track and the area surrounding it that is meant to be covered by sensing equipment will be called a “block”.

6.2. Problem analysis and solution

The scenario described in Section 6.1 is a particularly suitable example of a problem that can be approached with the use of autonomous agent systems. The mission is composed of several subtasks that are not necessarily sequential and require some sort of management system to engage or disable subsystems that performs them. In particular, the ASV is required to perform the following behaviours:

- If the mission area is only specified with a polygon, a *route planning* behaviour is required in order to generate waypoints to form the lawn mower path.

- *Path planning* is required to drive the vessel from one waypoint to the next, avoiding static and moving obstacles along the way.

- *Data processing* behaviours are required to analyse data coming from sensor such as cameras, lidar, radar, sonar and assess if possible mines are present, weather conditions, trajectory prediction for other vessels that may come within the range of the mission. Additionally surface coverage information can be inferred from positioning sensors such as Global Positioning System (GPS).

- *Hardware management* behaviours that convert instructions into actual commands for the actuators, in this case motors, rudder and communication devices.

These behaviors can be implemented as skills of the agent system. One could argue that this problem is also solvable with classical control. However the use of autonomous agent systems significantly simplifies the implementation of the logic, especially when using NLP languages such as sEnglish. Furthermore the situation described above implies the need for decision making capabilities that are difficult to achieve with classical control. For instance if the weather conditions reach a critical point, there are multiple equally valid options to be pondered about:

1. The vessel could interrupt the mission and fall back to a safe place. This would imply giving priority to the hardware itself over the mission.

2. The vessel could continue the mission risking the integrity of the hardware but keeping the probability of actually completing the mission to an acceptable level.

Again, this could be achieved with an intricate set of '`if`' statements, that could consider every single possibility. However BDI agents significantly simplify the approach to these problems by providing an architecture that allows to modularise and separate the logic, expressed in a human friendly language, from the lower level control.

Formal verification of the agent reasoning provides a way to verify that the logic reflects the original specifications, and that the agent will not try to perform specific actions that are intrinsically wrong or even dangerous for the hardware and for its surroundings.

Assuming probability distributions of environmental events are known, the LISA system is a great tool to facilitate the verification of the agent reasoning. When implementing the agent logic, the developer has the possibility of including probabilistic information within the agent program, which will then be used by the system to generate a model that can be verified, in this case with the probabilistic model checker PRISM.

6.3. LISA agent design

This section describes a simple approach that can be taken to solve the scenario of this case study with the LISA system and the framework described in this thesis.

This is clearly not a unique solution, and it is intended to be a proof of concept. The plan described in Section 6.1 lends itself well for use with the LISA system as it requires multiple lower level skills, some of them as a one time use and some of them as a continuous operation.

The surveying is organised as follows. Once a lawn mower surveying path is generated, the ASV will be driven to the starting point and it will start the survey. Each block is covered by going from a starting waypoint to an end waypoint, therefore the agent will need a percept that will tell it when a waypoint is reached, and one when the last waypoint is reached.

In this implementation two environmental conditions were considered:

- *Weather.* This kind of missions at open sea can be highly influenced by weather conditions, and it is reasonable to assume that even a small vessel possess sensing equipment able to determine the state of the weather, or at least communication devices that can fetch this information from elsewhere.
- *Coverage.* Given a map, the current pose in the environment and information provided by the sensing equipment, the system is able to assess, on the fly, whether or not the coverage of the surveyed area is complete. Since complete coverage is key in this scenario, the system will have to go back and cover spots that were left unclear.

This information leads in turn to two kind of events that can be generated to the agent reasoning. The first is when weather conditions change from normal to excessively harsh and vice versa. When the weather becomes too harsh the agent was designed to wait for instructions from human operators, reason being that in some cases it is hard to give a general rule as security situations may change and humans may still want to have the final decision, especially in military environments. The second environmental event that can happen in this situation is that when reaching a global waypoint there have been areas left unexplored in the last block. In this case the agent is faced with an important decision, that is whether to go back immediately and re-explore the spots that were missed, or keep going on with the next block and go back to re-explore at a later time. This decision could be implemented to be made based on arbitrary rules, possibly dependent on sensor readings, or it could be implemented as a nondeterministic decision that the agent is supposed to make.

Figure 6.2 shows the high level architecture for the LISA agent developed for this example. The diagram summarises to two kinds of data input: weather data and position data, which are not necessarily single streams of data but can be arrays of data coming from all sorts of different sensors. The data is fused and passed on to abstraction skills which are able to convert the information to sensory percepts for the agent reasoning to process. A path planner is present to convert high level instructions from the agent reasoning to sequences of waypoints, which will be followed in a safe way, avoiding collision with obstacles. A motion planner then converts waypoint goals into thrust and rudder command which are followed using PI control.

Appendix B.1 reports the full sEnglish reasoning code implemented for this case study. This agent is programmed to have four sensory percepts: '`Sea state is too high`', '`I am at global waypoint`', '`Areas left unexplored`' and '`Last waypoint reached`' (lines 9-12). Note that the program includes probability information that will be used by the translator to generate a probabilistic model in PRISM input language. Table 6.1 reports a list of all the available actions and related action feedbacks for this example. There are four actions and one of them features two possible action feedbacks. In this case the distributions for the action feedbacks were chosen arbitrarily, as a proof of concept rather than as an accurate modelling exercise.

Logic based implication rules are listed from line 22 to line 26. In this case they

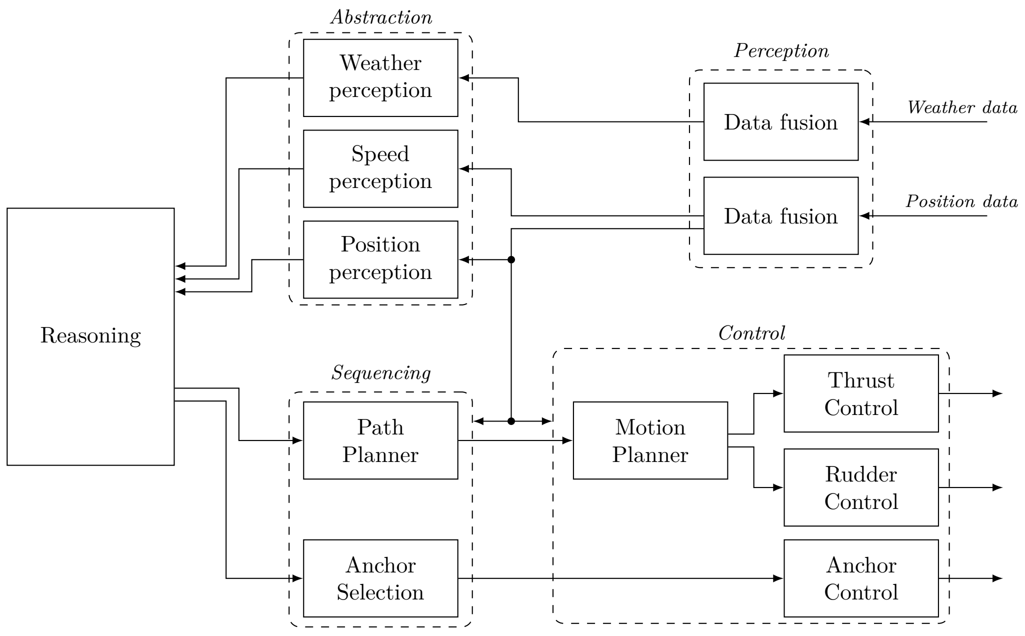


Figure 6.2.: *High level architecture and data flow of the LISA agent designed for the case study.*

Table 6.1.: *List of Actions and related Action feedbacks for the case study.*

Action	*Action feedback(s)*	*Distribution(s)*
`Generate set of waypoints`	`Waypoints generated`	$[1, 1, 0]$
`Activate drive mode`	`Drive mode`	$[1, 5, 0]$
`Activate park mode`	`Park mode`	$[1, 1, 0]$
`Wait for instructions`	`Continue`, `Abort`	$[0.6, 5, 2]$,$[0.4, 5, 2]$

were used to generate an '`Error`' belief when conflicting beliefs become true at the same time, for example if '`Drive mode`' and '`Park mode`' are activated simultaneously.

From line 28 the program lists all the executable plans. This implementation features 10 executable plans. Some of the plans are minimal and they only perform modification on mental notes. With the LISA system it would be possible to implement these operations as logic based implication rules, in the '`REASONING`' section, however in this case a larger number of plans was implemented as a proof of concept to show that model checking can still be performed in reasonable times with larger models. In particular Plan 4 (line 46) and Plan 5 (line 50) were implemented so to feature the same triggering condition, and test the process with a MDP implementation.

6.4. Verification

Appendix B.2 shows the PRISM model that was automatically generated from the agent code. Variables that represent beliefs and actions are named by copying the original atomic predicate, converting it completely to lowercase letters and substituting spaces with underscores.

For the purpose of comparing two similar models, a second implementation was created by implementing the agent logic in a very similar way but avoiding non-determinism in order to create a DTMC model. In Table 6.2 are reported results obtained by running the DTMC and MDP models in PRISM. All the tests were run on a Apple laptop with dual-core Intel Core i5-4258U 2.4GHz CPU, 16 GB of DDR3 memory, and running 64-bit Mac OS X 10.11.5.

As expected the MDP model generated a state space that is almost twice as large as the DTMC counterpart, and it uses almost three time as much memory as the DTMC model. This is probably due to the way PRISM builds models in memory: not

Table 6.2.: *Verification model building and results for* $\texttt{P}_{\texttt{min}=?}$ $[\texttt{F}_{\leq 100}\ \texttt{mission\_complete} = \texttt{1}]$

Model	**States**	**Transitions**	**Choices**	**Build time**	**Ver. time**	**Memory**	**Result**
MDP	270 268	420 431	276 454	38.061 s	1.901 s	10.0 MB	0.6357
DTMC	157 072	231 148	N/A	38.146 s	2.052 s	3.5 MB	0.6389

all states are stored but the software constructs a Multi-Terminal Binary Decision Diagram (MTBDD) [87] structures, an evolution of Binary Decision Diagram (BDD) [43], which are very much dependent on the type and structure of a model rather than the number of states.

Both models were then ran with a standard verification query that calculates the minimum probability of completing the mission within 100 steps. The verification engine used in PRISM is the *Sparse* engine which resulted in considerably faster performances compared to other engines.

The model construction time and the verification time resulted to be very similar for the two implementations. Note that both model construction time and verification time depend on many factors such as the computational speed of the machine, the resources allocated by the operating system and so on. The times reported in Table 6.2 are averaged over a sequence of runs in typical condition on the same machine.

An example of use for verification is to verify if conflicting actions are executed at the same time or particular predicates become true at the same time. For example the action '`Wait for instructions`' is defined with two possible action feedbacks: '`Continue`' or '`Abort`' (see Table 6.1). With this setup it is easily possible to verify that these two action feedbacks are never active at the same time with the specification $\texttt{P}_{=?}$ $[\texttt{F}\ (\texttt{abort} = \texttt{1}\ \&\ \texttt{continue} = \texttt{1})]$ for the DTMC and the specification $\texttt{P}_{\texttt{max}=?}$ $[\texttt{F}\ (\texttt{abort} = \texttt{1}\ \&\ \texttt{continue} = \texttt{1})]$ for the MDP. For both models the verification gives indeed a probability of 0, as reported in Table 6.3.

Another possible use of this tool is to verify what is the probability of the mission being complete without a particular event happening. For example in the agent code for the case study in Appendix B.1 logic based implication rules are defined to determine an error state. To verify the probability that the mission can be completed without the error variable becoming true, the following query can be run on the DTMC model: $\texttt{P}_{=?}\,[\texttt{F}\,(\texttt{error} = \texttt{0}\,\&\,\texttt{mission\_complete} = \texttt{1})]$ and $\texttt{P}_{\texttt{max}=?}\,[\texttt{F}\,(\texttt{error} = \texttt{0}\,\&\,\texttt{mission\_complete} = \texttt{1})]$ on the MDP model. The results reported in Table 6.3 indicate that under these conditions the probability of finishing the mission without errors are relatively low. This can be due to several factors, for example a fault in the logic could cause two conflicting predicates to be active at the same time and trigger the error condition.

Using the same error state, another example is to verify what is probability of an error to occur within a given number of steps. For example for 100 steps this is achieved with the query $\texttt{P}_{=?}\,[\texttt{F}_{\leq 100}\,\texttt{error} = \texttt{1}]$ for the DTMC model and $\texttt{P}_{\texttt{max}=?}\,[\texttt{F}_{\leq 100}\,\texttt{error} = \texttt{1}]$ for the MDP model. As for the previous example specification, the results in Table 6.3 indicate that there is likely a fault in the logic that allows two conflicting predicates to be triggered at the same time.

As mentioned before, PRISM allows to verify reward-based properties. For example on line 54 of the agent program in Appendix B.1 two reward values are associated with the internal action of adding the mental note '`Re_exploring areas`': '`fuel`' and '`time`'. A possible application of this is to verify what is the expected fuel consumption when the mission is complete. This is achieved in the DTMC with the query: $\texttt{R}\{\texttt{fuel}\}_{=?}\,[\texttt{F}\,\texttt{mission\_complete} = \texttt{1}]$ and in the MDP model with $\texttt{R}\{\texttt{fuel}\}_{\texttt{max}=?}\,[\texttt{F}\,\texttt{mission\_complete} = \texttt{1}]$.

Table 6.3 shows numerical results obtained by running all of the above-mentioned example specifications on the models automatically generated for this case study. The similarity between the results for the DTMC and the MDP model is an in-

Table 6.3.: *Verification results for different example specifications.*

Model	**Query**	**Result**
DTMC	$\texttt{P}_{=?}$ [$\texttt{F}$ ($\texttt{abort} = 1$ & $\texttt{continue} = 1$)]	0
MDP	$\texttt{P}_{\texttt{max}=?}$ [$\texttt{F}$ ($\texttt{abort} = 1$ & $\texttt{continue} = 1$)]	0
DTMC	$\texttt{P}_{=?}$ [$\texttt{F}$ ($\texttt{error} = 0$ & $\texttt{mission\_complete} = 1$)]	0.4416
MDP	$\texttt{P}_{\texttt{max}=?}$ [$\texttt{F}$ ($\texttt{error} = 0$ & $\texttt{mission\_complete} = 1$)]	0.4470
DTMC	$\texttt{P}_{=?}$ [$\texttt{F}_{\leq 100}$ $\texttt{error} = 1$]	0.6713
MDP	$\texttt{P}_{\texttt{max}=?}$ [$\texttt{F}_{\leq 100}$ $\texttt{error} = 1$]	0.6723
DTMC	$\texttt{R}\{\texttt{fuel}\}_{=?}$ [$\texttt{F}$ $\texttt{mission\_complete} = 1$]	5.7664
MDP	$\texttt{R}\{\texttt{fuel}\}_{\texttt{max}=?}$ [$\texttt{F}$ $\texttt{mission\_complete} = 1$]	5.7671

dication that the behaviour of the logic is very similar, suggesting that the MDP implementation in this case does not give any obvious performance advantage over the DTMC implementation.

6.5. Simulation

This section shows an example implementation of this case study in a simulation environment. For this example the approach taken was the direct implementation with MATLAB/SIMULINK as described in Subsection 5.2.2.

The structure of the Simulink model is that of Figure 2.1, with the agent reasoning implemented as a MATLAB function (see Figure 5.2) which controls a set of skills, and a dynamical model of the boat that represents the environment. In addition a Virtual Reality (VR) 3D model is connected to visualise the position of the boat relative to its surroundings.

The output of the agent reasoning is a vector of Boolean values that represent actions. These action variables activate *enabled* subsystems, that implement `runRepeated` skills, and *triggered* subsystems, that represent `runOnce` skills.

Figure 6.3 shows a partial diagram of the system, which includes the dynamical

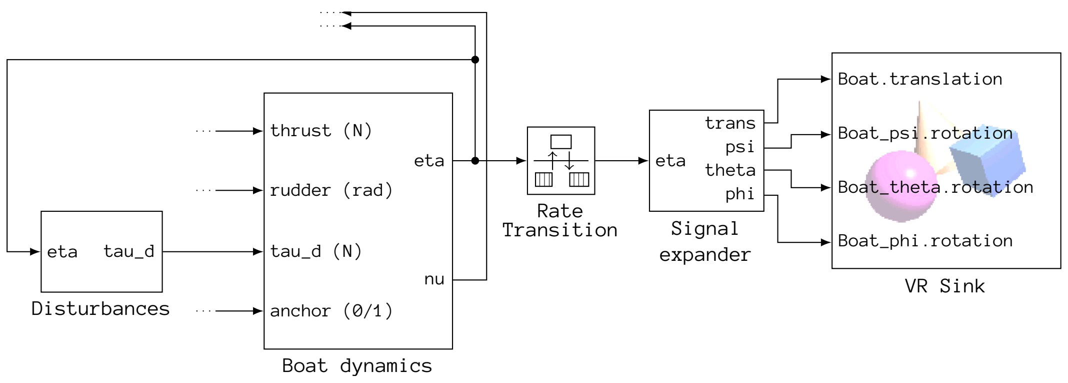


Figure 6.3.: *Simulink diagram of the simulated environment of a marine vessel with a Virtual Reality 3D visualiser.*

model of the boat and the 3D visualiser. A detailed description of the theory behind the dynamical model is given in Appendix A.1. The model was implemented making use of a 6-DOF equation of motion block from a MATLAB toolbox called Marine Systems Simulator (MSS) [155], with external forces manually implemented as described in Appendix A.1. External disturbances are calculated by applying a set of sinusoidal waves, calculating the forces that they generate along each axis. The model takes three inputs: the thrust generated by a motor positioned in the centre back of the boat, expressed in Newtons, the angle of the rudder expressed in radians and a boolean variable to control the state of a virtual anchor. The output that the model generates is a vector η with linear and angular positions, and a vector ν with linear and angular velocities. The linear positions and velocities are relative to a fixed frame centred in the middle of the map, the angular ones are relative to a body frame centred in the Centre Of Gravity (COG).

The `VR Sink` block of Figure 6.3 operates the virtual 3D world by loading a dedicated model file and changing its state with the inputs provided to it. Figure 6.4 shows a screenshot of the 3D world that visualises the linear and angular positions with a 3D rendering of the boat. The virtual environment was developed with the SIMULINK 3D Animation Toolbox.

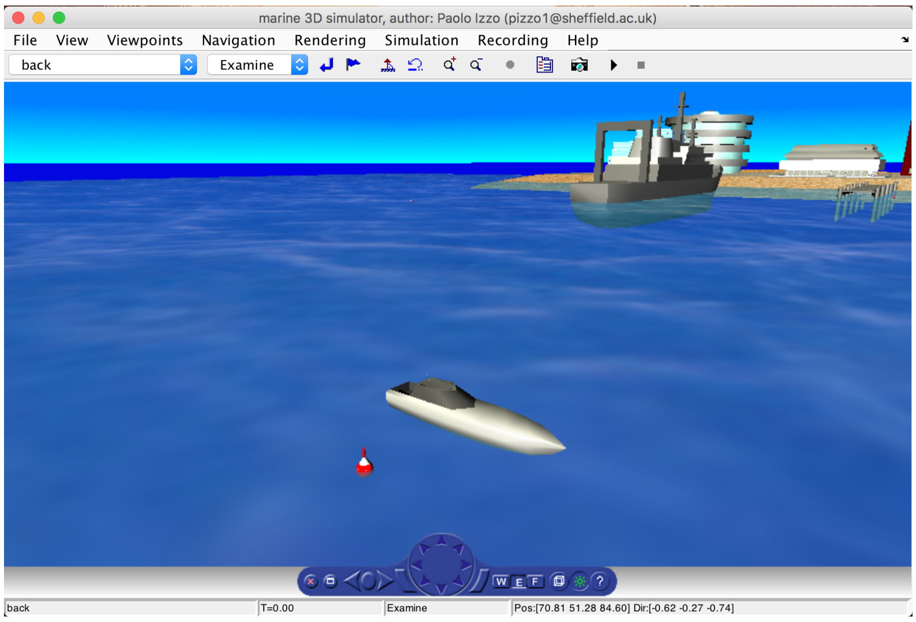

marine 3D simulator, author: Paolo Izzo (pizzo1@sheffield.ac.uk)
File View Viewpoints Navigation Rendering Simulation Recording Help
back
Examine
back
T=0.00
Examine
Pos:[70.81 51.28 84.60] Dir:[-0.62 -0.27 -0.74]

Figure 6.4.: *Screenshot of the Virtual Reality 3D environment of the vessel for the case study.*

6.5.1. Skills

For this case study a set of skills, in the form of MATLAB scripts, have been developed. They can be divided in three categories: *Control* skills, which regulate the thrust and rudder output to the boat, *Perception* skills, which generate inputs for the agent reasoning based on the simulation outputs, and *Planning* skills, which generate sets of waypoints for the boat to follow.

Table 6.4.: *List of the skills of the agent.*

Routine	**Subsystem**	**Description**
Initialisation	Control	Initialisation of global variables and some of the literals.

Table 6.4.: *List of the skills of the agent.*

Routine	**Subsystem**	**Description**
Percept process	Percept	Calculates the current absolute speed and tells the agent whether the boat is travelling at the desired speed. Calculates the distance from the next target and tells the agent if the boat is approaching the target or if it has reached it.
Path Planner	Planning	From the user input of destination and the current position of the boat, generates a sequence of waypoints that the boat will follow to avoid collision, using the algorithm described in [122].
Motion planner	Planning	Establishes a 'set speed' based on the user input and the current situation, and establishes a 'set target', which is the next waypoint to pursue.
Anchor on/off	Control	Simulates the behaviour of a real anchor by forcing the boat to hold the current position.
Move towards target	Control	Calculates the current target relative position and sets the rudder angle accordingly. Sets the thrust according to the 'set speed' which is an input of the routine, it applies PI control to eliminate steady state error.

6.6. Conclusions

This chapter described the application of the agent-based system LISA to a case study. The scenario described was that of a small marine vessel, an ASV, on a mission for mine detection and disposal in an environment delimited by a closed sequence of coordinates. The mission for the ASV is to explore a given area, completely covering the area with the range given by the onboard sensing equipment, and tag potential treat that will later on be considered by human operators in a nearby control centre.

An example agent reasoning was developed in the LISA framework to solve the case study scenario. The agent controls the vessel to explore the unknown area in a lawn mower path, by dividing the area into blocks. Two environmental conditions were considered: weather changes, to account for possible safety concerns when the weather condition becomes too harsh, and coverage control, to account for spots left unclear when covering blocks. Changes in these environmental conditions trigger plans that implement safety procedures, in case of harsh weather, and re-exploring manoeuvres in case there are spots left unclear.

The example agent reasoning was implemented with two approaches, one to be converted to a MDP and one to be converted to a DTMC. Both probabilistic models where built with the model checker PRISM, and they were both first verified with a standard verification query that checks what is the minimum probability of completing the mission within 100 steps, and then with a series of additional queries that tested the efficacy of the logic and illustrated how verification can be used to analyse the agent program. The models generated in PRISM were reasonably sized for such an example with model construction times for both the MDP and the DTMC model, on average, under 40 seconds. The MDP model, as expected, produced a structure with a much larger state space compared to the DTMC implementation. However verification results suggested that the MDP implementation does not provide any immediate advantage over the DTMC implementation.

A simulation environment for the case study was also implemented with MATLAB/SIMULINK. The agent reasoning was implemented as a MATLAB function, which activate skills also implemented as MATLAB functions. A 3D VR environment was implemented in SIMULINK to visualise the output of a dynamical model of the vessel. The dynamical model considered is a 6DOF rigid-body dynamical model which takes into account a set of external forces such as skin friction and drag forces.

Chapter 7.

Conclusions

7.1. Summary

This thesis described a novel agent architecture, called the Limited Instruction Set Agent (LISA) system, which features an agent reasoning that can be automatically verified by model checking. The idea is to enable the user to include known probabilistic information in the program that describes the agent logic, allowing for the implementation of a system that automatically generates a probabilistic model that can be verified with known probabilistic verification tools.

The literature review formally defined rational agents and agent oriented programming. Existing work on autonomous agent verification was described, highlighting possible drawbacks that can be addressed in future development. Definitions of formal verification and model checking were presented, with particular attention to the models used in this project: Discrete-Time Markov Chain (DTMC) and Markov Decision Process (MDP). A brief overview of algorithms that can be used as skills of the agent was also given.

Amongst the numerous papers on the topic of verification of autonomous agents, recurring problems were found. For instance the probabilistic nature of sensory percepts of agents is rarely taken into account. Another problem was that the creation of a model that can be verified by a model checking software is often performed by executing a symbolic model of the agent program to explore every reachable path, and eventually list every reachable state in the state space, making the process highly computationally expensive.

The LISA system aims to address these problems by creating a framework that is structurally simpler and it facilitates verification by model checking. The architecture is based on BDI and three layer architectures, with the agent reasoning on top and two lower levels of subsystems that the agent reasoning can control and it uses as an interface with the external world. The agent reasoning is based on Jason, an evolution of AgentSpeak. Modifications are made to Jason so to facilitate modelling and verification, as well as reducing the size of the state space required to build the model.

The agent program is defined in the NLP language sEnglish, which is enriched with structures that allow to introduce a probabilistic model of environmental events within the agent code. This, in turn, allows to automatically generate a probabilistic model of the agent reasoning directly from the agent code. LISA reasoning was shown to abstract away as a DTMC in the particular case when plans have unique triggering conditions, and to always be abstract as a MDP. The approach proposed here to formal verification of agent systems still requires the user to define the probability distributions to describe environmental events, however, it represents a tool that allows to easily implement a verification process when probability distributions are known.

The software chosen for probabilistic verification was Prism. The model generated from the agent code was shown to be useful for both design-time and run-time verification. Design-time verification can be used to improve and validate the agent design for autonomous control. Run-time verification can be used to improve the decision-making capabilities of the control agent by implementing model checking techniques in realtime as a means of internal model-based simulation, in order to predict outcomes of actions and choose the most suitable strategy.

Finally, a set of possibilities for the implementation and simulation of the LISA system was presented. In particular the LISA system was shown to be implementable with the Cognitive Agent Toolbox (CAT), with MATLAB/SIMULINK as well as in popular robotic packages such as ROS and MOOS-IvP.

7.2. Results and challenges

The capabilities of the LISA framework were demonstrated with a case study. The scenario presented is that of an Autonomous Surface Vehicle (ASV) for mine detection and disposal.

The logic behind the implementation of the agent reasoning was explained and verification results were presented. Agent reasoning was implemented using two different approaches, one to generate a MDP model and one to generate a DTMC model. A possible simulation environment for the agent system was also presented, with 6-DOF dynamical model and a VR 3D visual environment.

The case study showed promising results. Even though the logic behind this particular example is fairly complex, the model building times for both the DTMC and the MDP implementations of the model resulted to be of around 40 s, and the model checking times of about 2 s for the example of specification that was used.

Some challenges were found during the development of the system. In particular, increasing the number or the length of plans, or introducing new mental notes, results in a larger number of states required for the discrete model in PRISM, which in turn results in larger model building and verification times. Unless there is a disproportionate increase of the state space, this is hardly a problem for design-time verification, especially when there are no set limitations for energy consumption, size, weight and so on. However, in the case of run-time verification, the increase in verification time can potentially brake the applicability of the concept. For example for the Plan Selection Function implementation, run-time verification would need to be executed within a single reasoning cycle, unless there is a mechanism in place for suspending the choice until the verification process is done. In case of implementation as an external skill, the timing window of the run-time verification can be generally wider, assuming that the beliefs associated with it are not required to be updated frequently.

7.3. Future Work

The LISA system presented in this thesis shows a great potential for application in a multitude of scenarios. There are however some points to be addressed in future implementations.

The run-time verification framework described in Section 4.5, aside from preliminary testing, has not been extensively tested in real world applications. It would be interesting to see how this process scales up with increasingly complex reasoning agents, and how differently it performs on machines with different levels of computational power.

In Jason and other similar AgentSpeak-derived languages, beliefs are given an additional degree of abstraction over the true/false status of boolean variables. Agent reasoning can believe that something is *true* or *false* but also *not true* or *not false*. This is achieved by looking at the Beliefs set for missing information about one or more beliefs. In the LISA implementation of the agent reasoning, this process is still in place. However when automatically generating the probabilistic model of the agent reasoning, this possibility has not yet been explored. Future implementation of the translation and abstraction script could take into account this abstraction, by giving multiple states to belief variables. However it is possible that this would greatly increase the size of the state space if a large number of these states are reachable during the operation of the model.

Another interesting aspect that could be addressed in future implementation is the application of the concept of automatic probabilistic modelling and verification to multi-agent system [69, 129]. For example if multiple agents are implemented with the LISA framework, a mechanism for automatically generate a model that encapsulate the behaviour of the group could be created, giving the means of predicting the outcome of certain behaviours and interactions.

Appendix A.

Additional material

A.1. Dynamics of marine vessels

Consider a rigid body that models the structure of a small marine vessel. Although every rigid body presents some flexibility to a certain extent, the model described here considers the vessel to be a completely rigid body. Rigid body dynamics can be described in a three dimensional space with the 6-DOF equations of motion, which decomposes the dynamics along three translational axes and three rotational axes. Figure A.1 shows a representation of the reference frames for the dynamical model. A frame is fixed to the rigid body and centred in the COG, called the *b-frame*, and a frame is fixed to the ground, called the *n-frame*.

Given a *b-frame* and an *n-frame* as described in Figure A.1, the 6-DOF equations of motion can be defined as follows [83, 84]:

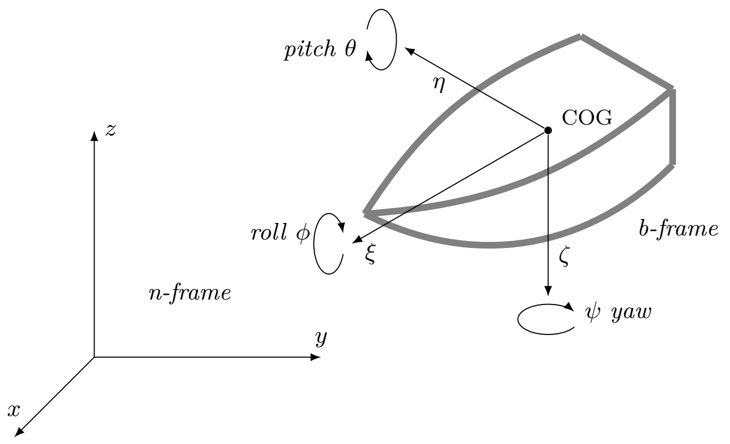


Figure A.1.: *Reference frames for the dynamical model.*

$$\begin{cases}
m[\dot{u} - vr + wq] &= R_\xi \\
m[\dot{v} - wp + ur] &= R_\eta \\
m[\dot{w} - uq + vp] &= R_\zeta \\
I_\xi \dot{p} + (I_\zeta - I_\eta)qr - (\dot{r} + pq)I_{\xi\zeta} + (r^2 - q^2)I_{\eta\zeta} + (pr - \dot{q})I_{\xi\eta} &= M_\phi \\
I_\eta \dot{q} + (I_\xi - I_\zeta)rp - (\dot{p} + qr)I_{\xi\eta} + (p^2 - r^2)I_{\zeta\xi} + (qp - \dot{r})I_{\eta\zeta} &= M_\theta \\
I_\zeta \dot{r} + (I_\eta - I_\xi)pq - (\dot{q} + rp)I_{\eta\zeta} + (q^2 - p^2)I_{\xi\eta} + (rq - \dot{p})I_{\zeta\xi} &= M_\psi
\end{cases} \tag{A.1}$$

where:

	m	total mass of the boat
$\boldsymbol{\eta} =$	$[x\ y\ z\ \phi\ \theta\ \psi]^T$	Position vector of the b-frame relative to the n-frame.
$\boldsymbol{\nu} =$	$[u\ v\ w\ p\ q\ r]^T$	Linear and angular velocities decomposed in the b-frame.
$\boldsymbol{\tau} =$	$[R_\xi\ R_\eta\ R_\zeta\ M_\phi\ M_\theta\ M_\psi]^T$	External forces and momentums decomposed in the b-frame.

$$I_0 = \begin{bmatrix} I_\xi & I_{\xi\eta} & I_{\xi\zeta} \\ I_{\eta\xi} & I_\eta & I_{\eta\zeta} \\ I_{\zeta\xi} & I_{\zeta\eta} & I_\zeta \end{bmatrix} \qquad \text{Inertia tensor}$$

By approximating the rigid body as a system of M distributed masses, the components of the Inertia tensor I_0 can be defined as follows.

$$\begin{array}{ll} I_\xi = \sum_{i=1}^{M} (\eta_i^2 + \zeta_i^2)\, m_i & I_{\xi\eta} = I_{\eta\xi} = -\sum_{i=1}^{M} \xi_i\, \eta_i\, m_i \\ I_\eta = \sum_{i=1}^{M} (\xi_i^2 + \zeta_i^2)\, m_i & I_{\eta\zeta} = I_{\zeta\eta} = -\sum_{i=1}^{M} \eta_i\, \zeta_i\, m_i \\ I_\zeta = \sum_{i=1}^{M} (\xi_i^2 + \eta_i^2)\, m_i & I_{\xi\zeta} = I_{\zeta\xi} = -\sum_{i=1}^{M} \xi_i\, \zeta_i\, m_i \end{array} \tag{A.2}$$

where $[\xi_i, \eta_i, \zeta_i]$ is the position of the i-th mass m_i relative to the COG.

The external forces and momentums $\boldsymbol{\tau}$ can be identified as the sum of three components: *hydrodynamic* forces and momentums $\boldsymbol{\tau_H}$, *propulsion* forces and momentums $\boldsymbol{\tau_P}$ and *environmental* (disturbance) forces and momentums τ_d:

$$\boldsymbol{\tau} = \boldsymbol{\tau_H} + \boldsymbol{\tau_P} + \boldsymbol{\tau_d} \tag{A.3}$$

Hydrodynamic forces in turn can be expressed as the sum of *added mass*, *hydrodynamic damping* and *restoring forces*:

$$\boldsymbol{\tau}_H = -\underbrace{[M_A\dot{\boldsymbol{\nu}} + C_A(\boldsymbol{\nu})\boldsymbol{\nu}]}_{\text{added mass}} - \underbrace{D(\boldsymbol{\nu})\boldsymbol{\nu}}_{\substack{\text{hydrodynamic}\\ \text{damping}}} - \underbrace{\boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0}_{\text{restoring forces}} \tag{A.4}$$

Added mass (also known as *virtual mass*) is the force generated by the volume of water surrounding the vehicle as it moves through it. *Hydrodynamic* damping forces can include skin friction, wave drift damping and damping due to vortex shedding. *Restoring* forces are due to Archimedes law (weight and buoyancy).

In this implementation the external forces and momentums where implemented as follows:

$$
\begin{aligned}
R_\xi &= -K_\xi u^2 \operatorname{sign}(u) + R_T + R_{\xi d} \\
R_\eta &= -K_\eta v^2 \operatorname{sign}(v) + R_{\eta d} \\
R_\zeta &= -K_\zeta w^2 \operatorname{sign}(w) - \bar{V}(z)\rho_w g + mg + R_{\zeta d} \\
M_\phi &= -K_{\phi_1} p - K_{\phi_2} p^2 \operatorname{sign}(p) - K_{\phi_3} sin(\phi) + M_{\phi d} \\
M_\theta &= -K_{\theta_1} q - K_{\theta_2} q^2 \operatorname{sign}(q) - K_{\theta_3} sin(\theta) + M_{\theta d} \\
M_\psi &= -K_{\eta\psi} v^2 \operatorname{sign}(v) - K_\psi r^2 \operatorname{sign}(r) + M_\delta + M_{\psi d}
\end{aligned}
\tag{A.5}
$$

where:

R_T	Thrust force (linear) along the x axis
$\bar{V}(z)$	Submerged volume of the boat
ρ_w	Water density
M_δ	Momentum generated by the rudder
δ	Rudder angle
m	Total mass of the boat
g	Gravity acceleration

The terms denoted with a '*d*' subscript represent the components of the environmental forces vector

$$
\boldsymbol{\tau}_d = \begin{bmatrix} R_{\xi d} & R_{\eta d} & R_{\zeta d} & M_{\phi d} & M_{\theta d} & M_{\psi d} \end{bmatrix}^T \tag{A.6}
$$

The *propulsion* term $\boldsymbol{\tau}_P$, which represents the control input, is composed by the *thrust force* R_T, expressed in Newtons and linear to the ξ axis, and the momentum generated by the action of the *rudder* on the yaw, denoted with M_δ, expressed as follows:

$$M_\delta = K_T u^2 \operatorname{sign}(u) \cos(\delta) \sin(\delta)$$

where δ is the angular position of the rudder with respect to the ξ axis.

The hydrodynamic forces make up the rest of the terms in Equation A.5. The added mass forces and momentum are considered to be negligible in this implementation, with the assumption that a small vessel features a reasonably aerodynamic profile and it does not travel at speeds that are high enough to generate a significant displacement of water. For the hydrodynamic damping, two main factors were considered: skin friction and drag. Skin friction terms are present on the roll and pitch axes, and they are proportional to the respective velocities along those axes. Drag terms are present on all axes and proportional to the velocity squared along the relative axes. Drag forces can be expressed with the following formula [15]:

$$F_D = \frac{1}{2}\rho \bar{v}^2 C_D A \tag{A.7}$$

where $\bar{v}$ is the velocity of the vehicle along the axis in question. The mass density of the fluid ρ, the drag coefficient C_D and the area A of the vehicle facing the fluid during motion are constants that are summarised under the 'K' terms in Equation A.5. Finally the following *restoring forces* are considered: one linear to the vertical axis ζ, one for the roll angle and one for the pitch angle. The vertical restoring force is the sum of two opposing forces: the buoyancy force, proportional to the submerged volume of the boat times the gravity acceleration, and the weight force. The restoring forces along the roll and pitch angles are proportional to the sine of the respective angles.

Appendix B.

Code for the case study

B.1. Agent reasoning code

```
INITIAL BELIEFS AND GOALS
Start mission.

INITIAL ACTIONS

PERCEPTION PROCESS
Monitor the following booleans:
//Percepts
Sea state is too high. {[],[0.1,10,0],[0.5,3,0]}
I am at global waypoint. {[],[0.1,10,0],[0.5,3,0]}
Areas left unexplored. {[I am at global waypoint
    ],[0.1,1,0],[0.1,1,0]}
Last waypoint reached. {[I am at global waypoint
    ],[0.1,1,0],[0.1,1,0]}
//Action feedbacks
Waypoints generated.
Drive mode.
Park mode.
Continue.
Abort.

Monitor the following objects:

REASONING
If ^[Abort] and ^[Continue] then ^[Error]
If ^[Last waypoint reached] and ~^[I am at global waypoint] then ^[
    Error]
If ^[Drive mode] and ^[Park mode] then ^[Error]
If ^[I am not going back] and ^[Re_exploring_areas] then ~^[I am not
     going back]

EXECUTABLE PLANS
```

```
//Plan 1
If ^[Start mission] while ~^[Sea state is too high] then
  [Generate set of waypoints.]
  [Activate drive mode.]
  +^[Exploring block]
  -^[Start mission].

//Plan 2
If ^[I am at global waypoint] while ^[Exploring block] then
  -^[Exploring block]
  +^[Block explored]. {fuel=1,time=1}

//Plan 3
If ~^[I am at global waypoint] while ^[Drive mode] then
  -^[Block explored]
  +^[Exploring block].

//Plan 4
If ^[Block explored] while ^[Areas left unexplored] and ~^[Sea state
     is too high] then
  +^[I am not going back].

//Plan 5
If ^[Block explored] while ^[Areas left unexplored] and ~^[Sea state
     is too high] then
  [Activate park mode.]
  [Generate set of waypoints.]
  +^[Re_exploring areas] {fuel=1,time=1}
  [Activate drive mode.].

//Plan 6
If ^[Block explored] while ^[Re_exploring areas] then
  -^[Re_exploring areas].

//Plan 7
If ^[Last waypoint reached] while ^[Block explored] and ~^[Areas
    left unexplored] then
  [Activate park mode.]
  +^[Mission complete].

//Plan 8
If ^[Sea state is too high] while true then
  [Activate park mode.]
  [Wait for instructions.] {time=1}
  +^[Waiting for instructions].

//Plan 9
If ^[Continue] while ^[Waiting for instructions] then
  [Activate drive mode.]
  -^[Waiting for instructions].

//Plan 10
If ^[Abort] while true then
```

```
  [Activate park mode.].
```

B.2. Automatically generated PRISM code of the MDP model

```
mdp

// ***** PLANS ***** //

module plan_1
plan_1: [0..4] init 0;

[t] plan_1=0 & !(start_mission=1 & (sea_state_is_too_high=0)) -> (plan_1'=0);
[t] plan_1=0 & (start_mission=1 & (sea_state_is_too_high=0)) -> (plan_1'=1);
//generate_set_of_waypoints
[t] plan_1=1 & !(waypoints_generated=1) -> (plan_1'=1);
[t] plan_1=1 & (waypoints_generated=1) -> (plan_1'=2);
//activate_drive_mode
[t] plan_1=2 & !(drive_mode=1) -> (plan_1'=2);
[t] plan_1=2 & (drive_mode=1) -> (plan_1'=3);
//+exploring_block
[t] plan_1=3 -> (plan_1'=4);
//-start_mission
[t] plan_1=4 -> (plan_1'=0);
endmodule

module plan_2
plan_2: [0..2] init 0;

[t] plan_2=0 & !(i_am_at_global_waypoint=1 & (exploring_block=1)) -> (plan_2'=0);
[t] plan_2=0 & (i_am_at_global_waypoint=1 & (exploring_block=1)) -> (plan_2'=1);
//-exploring_block
[t] plan_2=1 -> (plan_2'=2);
//+block_explored
[t] plan_2=2 -> (plan_2'=0);
endmodule

module plan_3
plan_3: [0..2] init 0;

[t] plan_3=0 & !(i_am_at_global_waypoint=0 & (drive_mode=1)) -> (plan_3'=0);
[t] plan_3=0 & (i_am_at_global_waypoint=0 & (drive_mode=1)) -> (plan_3'=1);
//-block_explored
[t] plan_3=1 -> (plan_3'=2);
//+exploring_block
[t] plan_3=2 -> (plan_3'=0);
```

```
endmodule

module plan_4_5
plan_4: [0..1] init 0;
plan_5: [0..4] init 0;

//Triggering event
[t] (plan_4=0 & plan_5=0) & !(block_explored=1 & (areas_left_unexplored=1 & sea_state_is_too_high=0)) -> (plan_4'=0) & (plan_5'=0);
[t] (plan_4=0 & plan_5=0) & (block_explored=1 & (areas_left_unexplored=1 & sea_state_is_too_high=0)) -> (plan_4'=1);
[t] (plan_4=0 & plan_5=0) & (block_explored=1 & (areas_left_unexplored=1 & sea_state_is_too_high=0)) -> (plan_5'=1);

//Plan 4 actions
//+i_am_not_going_back
[t] plan_4=1 -> (plan_4'=0);

//Plan 5 actions
//activate_park_mode
[t] plan_5=1 & !(park_mode=1) -> (plan_5'=1);
[t] plan_5=1 & (park_mode=1) -> (plan_5'=2);
//generate_set_of_waypoints
[t] plan_5=2 & !(waypoints_generated=1) -> (plan_5'=2);
[t] plan_5=2 & (waypoints_generated=1) -> (plan_5'=3);
//+re_exploring_areas
[t] plan_5=3 -> (plan_5'=4);
//activate_drive_mode
[t] plan_5=4 & !(drive_mode=1) -> (plan_5'=4);
[t] plan_5=4 & (drive_mode=1) -> (plan_5'=0);
endmodule

module plan_6
plan_6: [0..1] init 0;

[t] plan_6=0 & !(block_explored=1 & (re_exploring_areas=1)) -> (plan_6'=0);
[t] plan_6=0 & (block_explored=1 & (re_exploring_areas=1)) -> (plan_6'=1);
//-re_exploring_areas
[t] plan_6=1 -> (plan_6'=0);
endmodule

module plan_7
plan_7: [0..2] init 0;

[t] plan_7=0 & !(last_waypoint_reached=1 & (block_explored=1 & areas_left_unexplored=0)) -> (plan_7'=0);
[t] plan_7=0 & (last_waypoint_reached=1 & (block_explored=1 & areas_left_unexplored=0)) -> (plan_7'=1);
```

```
//activate_park_mode
[t] plan_7=1 & !(park_mode=1) -> (plan_7'=1);
[t] plan_7=1 & (park_mode=1) -> (plan_7'=2);
//+mission_complete
[t] plan_7=2 -> (plan_7'=0);
endmodule

module plan_8
plan_8: [0..3] init 0;

[t] plan_8=0 & !(sea_state_is_too_high=1) -> (plan_8'=0);
[t] plan_8=0 & (sea_state_is_too_high=1) -> (plan_8'=1);
//activate_park_mode
[t] plan_8=1 & !(park_mode=1) -> (plan_8'=1);
[t] plan_8=1 & (park_mode=1) -> (plan_8'=2);
//wait_for_instructions
[t] plan_8=2 & !(continue=1 | abort=1) -> (plan_8'=2);
[t] plan_8=2 & (continue=1 | abort=1) -> (plan_8'=3);
//+waiting_for_instructions
[t] plan_8=3 -> (plan_8'=0);
endmodule

module plan_9
plan_9: [0..2] init 0;

[t] plan_9=0 & !(continue=1 & (waiting_for_instructions=1)) -> (
    plan_9'=0);
[t] plan_9=0 & (continue=1 & (waiting_for_instructions=1)) -> (
    plan_9'=1);
//activate_drive_mode
[t] plan_9=1 & !(drive_mode=1) -> (plan_9'=1);
[t] plan_9=1 & (drive_mode=1) -> (plan_9'=2);
//-waiting_for_instructions
[t] plan_9=2 -> (plan_9'=0);
endmodule

module plan_10
plan_10: [0..1] init 0;

[t] plan_10=0 & !(abort=1) -> (plan_10'=0);
[t] plan_10=0 & (abort=1) -> (plan_10'=1);
//activate_park_mode
[t] plan_10=1 & !(park_mode=1) -> (plan_10'=1);
[t] plan_10=1 & (park_mode=1) -> (plan_10'=0);
endmodule

// ***** ACTIONS ***** //

module activate_drive_mode
drive_mode: [0..5] init 0;
//drive_mode[1,5,0]
[b] !(plan_1=2 | plan_5=4 | plan_9=1) & (drive_mode<=1) -> (
    drive_mode'=0);
```

```
[b] (plan_1=2 | plan_5=4 | plan_9=1) & (drive_mode<=1) -> (
    drive_mode'=5);
[b] drive_mode>1 -> (drive_mode'=drive_mode-1);
endmodule

module generate_set_of_waypoints
waypoints_generated: [0..1] init 0;
//waypoints_generated[1,1,0]
[b] !(plan_1=1 | plan_5=2) & (waypoints_generated<=1) -> (
    waypoints_generated'=0);
[b] (plan_1=1 | plan_5=2) & (waypoints_generated<=1) -> (
    waypoints_generated'=1);
endmodule

module activate_park_mode
park_mode: [0..1] init 0;
//park_mode[1,1,0]
[b] !(plan_5=1 | plan_7=1 | plan_8=1 | plan_10=1) & (park_mode<=1)
    -> (park_mode'=0);
[b] (plan_5=1 | plan_7=1 | plan_8=1 | plan_10=1) & (park_mode<=1) ->
     (park_mode'=1);
endmodule

module wait_for_instructions
continue: [0..5] init 0;
abort: [0..1] init 0;
//continue[0.6,5,0] abort[0.4,5,0]
[b] !(plan_8=2) & (continue<=1 & abort<=1) -> (continue'=0) & (abort
    '=0);
[b] (plan_8=2) & (continue<=1 & abort<=1) -> (continue'=5);
[b] continue>2 -> (continue'=continue-1);
[b] continue=2 -> 0.6 : (continue'=1) & (abort'=0) + 0.4 : (continue
    '=0) & (abort'=1);
endmodule

// ***** MENTAL NOTES ***** //

module start_mission
start_mission: [0..1] init 1;
[b] start_mission=0 -> (start_mission'=0);
[b] start_mission=1 & (plan_1=4) -> (start_mission'=0);
[b] start_mission=1 & !(plan_1=4) -> (start_mission'=1);
endmodule

module error
error: [0..1] init 0;
[b] ((abort=1 & continue=1) | (last_waypoint_reached=1 &
    i_am_at_global_waypoint=0) | (drive_mode=1 & park_mode=1)) -> (
    error'=1);
[b] !((abort=1 & continue=1) | (last_waypoint_reached=1 &
    i_am_at_global_waypoint=0) | (drive_mode=1 & park_mode=1)) -> (
    error'=error);
endmodule
```

```
module exploring_block
exploring_block: [0..1] init 0;
[b] exploring_block=0 & (plan_1=3 | plan_3=2) -> (exploring_block
    '=1);
[b] exploring_block=0 & !(plan_1=3 | plan_3=2) -> (exploring_block
    '=0);
[b] exploring_block=1 & (plan_2=1) -> (exploring_block'=0);
[b] exploring_block=1 & !(plan_2=1) -> (exploring_block'=1);
endmodule

module block_explored
block_explored: [0..1] init 0;
[b] block_explored=0 & (plan_2=2) -> (block_explored'=1);
[b] block_explored=0 & !(plan_2=2) -> (block_explored'=0);
[b] block_explored=1 & (plan_3=1) -> (block_explored'=0);
[b] block_explored=1 & !(plan_3=1) -> (block_explored'=1);
endmodule

module i_am_not_going_back
i_am_not_going_back: [0..1] init 0;

[b] i_am_not_going_back=0 & ((plan_4=1) & !(re_exploring_areas=1 &
    i_am_not_going_back=1)) -> (i_am_not_going_back'=1);
[b] i_am_not_going_back=0 & !((plan_4=1) & !(re_exploring_areas=1 &
    i_am_not_going_back=1)) -> (i_am_not_going_back'=0);
[b] i_am_not_going_back=1 & (re_exploring_areas=1 &
    i_am_not_going_back=1) -> (i_am_not_going_back'=0);
[b] i_am_not_going_back=1 & !(re_exploring_areas=1 &
    i_am_not_going_back=1) -> (i_am_not_going_back'=1);

endmodule

module re_exploring_areas
re_exploring_areas: [0..1] init 0;
[b] re_exploring_areas=0 & (plan_5=3) -> (re_exploring_areas'=1);
[b] re_exploring_areas=0 & !(plan_5=3) -> (re_exploring_areas'=0);
[b] re_exploring_areas=1 & (plan_6=1) -> (re_exploring_areas'=0);
[b] re_exploring_areas=1 & !(plan_6=1) -> (re_exploring_areas'=1);
endmodule

module mission_complete
mission_complete: [0..1] init 0;
[b] mission_complete=0 & (plan_7=2) -> (mission_complete'=1);
[b] mission_complete=0 & !(plan_7=2) -> (mission_complete'=0);
[b] mission_complete=1 -> (mission_complete'=1);
endmodule

module waiting_for_instructions
waiting_for_instructions: [0..1] init 0;
[b] waiting_for_instructions=0 & (plan_8=3) -> (
    waiting_for_instructions'=1);
[b] waiting_for_instructions=0 & !(plan_8=3) -> (
    waiting_for_instructions'=0);
[b] waiting_for_instructions=1 & (plan_9=2) -> (
    waiting_for_instructions'=0);
[b] waiting_for_instructions=1 & !(plan_9=2) -> (
    waiting_for_instructions'=1);
endmodule

// ***** PERCEPTS ***** //

module sea_state_is_too_high
sea_state_is_too_high: [0..1] init 0;
c1: [0..10] init 1;

//[0.1,10,0]
[b] sea_state_is_too_high=0 & c1<10 -> (c1'=c1+1);
[b] sea_state_is_too_high=0 & c1>=10 -> 0.01 : (
    sea_state_is_too_high'=1) & (c1'=0) + 0.99 : (c1'=0);
//[0.5,3,0]
[b] sea_state_is_too_high=1 & c1<3 -> (c1'=c1+1);
[b] sea_state_is_too_high=1 & c1>=3 -> 0.5 : (sea_state_is_too_high
    '=0) & (c1'=0) + (1-0.5) : (c1'=0);
endmodule

module i_am_at_global_waypoint
i_am_at_global_waypoint: [0..1] init 0;
c2: [0..10] init 1;

//[0.1,10,0]
[b] i_am_at_global_waypoint=0 & c2<10 -> (c2'=c2+1);
[b] i_am_at_global_waypoint=0 & c2>=10 -> 0.9 : (
    i_am_at_global_waypoint'=1) & (c2'=0) + 0.1 : (c2'=0);
//[0.5,3,0]
[b] i_am_at_global_waypoint=1 & c2<3 -> (c2'=c2+1);
[b] i_am_at_global_waypoint=1 & c2>=3 -> (i_am_at_global_waypoint
    '=0) & (c2'=0);
endmodule

module areas_left_unexplored
areas_left_unexplored: [0..1] init 0;

[b] areas_left_unexplored=1 -> 0.1 : (areas_left_unexplored'=0) +
    0.9 : (areas_left_unexplored'=1);
[b] areas_left_unexplored=0 & i_am_at_global_waypoint=0 -> (
    areas_left_unexplored'=0);
[b] areas_left_unexplored=0 & i_am_at_global_waypoint=1 -> 0.1 : (
    areas_left_unexplored'=1) + 0.9 : (areas_left_unexplored'=0);
endmodule

module last_waypoint_reached
last_waypoint_reached: [0..1] init 0;

[b] last_waypoint_reached=1 -> (last_waypoint_reached'=1);
[b] last_waypoint_reached=0 & i_am_at_global_waypoint=0 -> (
    last_waypoint_reached'=0);
[b] last_waypoint_reached=0 & i_am_at_global_waypoint=1 -> 0.1 : (
    last_waypoint_reached'=1) + 0.9 : (last_waypoint_reached'=0);
endmodule

module scheduler
x: [0..1] init 0;

[b] x=0 -> (x'=1);
[t] x=1 -> (x'=0);
endmodule

rewards "res_cycles"
x=0 : 1;
endrewards

rewards "fuel"
plan_2=1 : 1;
plan_5=3 : 1;
endrewards

rewards "time"
plan_2=1 : 1;
plan_5=3 : 1;
plan_8=2 : 1;
endrewards
```

Bibliography

[1] P. E. Agre and D. Chapman, “Pengi: An implementation of a theory of activity.”, in *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, ser. AAAI’87, Seattle, WA, USA: AAAI Press, 1987, pp. 286–272.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab and F. Ingrand, “An architecture for autonomy”, *The International Journal of Robotics Research*, 17, no., pp. 315–337, 1998.

[3] R. D. Alexander, M. Hall-May and T. P. Kelly, “Certification of autonomous systems under uk military safety standards”, University of York, York, UK, Tech. Rep., 2007.

[4] R. Alur, T. A. Henzinger, G. Lafferriere and G. J. Pappas, “Discrete abstractions of hybrid systems”, *Proceedings of the IEEE*, 88, no., pp. 971–984, 2000.

[5] P. J. Antsaklis, K. M. Passino and S. J. Wang, “An introduction to autonomous control systems”, *IEEE Control Systems*, 11, no., pp. 15–13, 1991.

[6] P. J. Antsaklis, J. A. Stiver and M. Lemmon, “Hybrid system modeling and autonomous control systems”, in *Hybrid Systems*, R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel, Eds., Berlin, Heidelberg: Springer, 1993, pp. 366–392.

[7] H. Ardiny, S. Witwicki and F. Mondada, “Construction automation with autonomous mobile robots: A review”, in *Robotics and Mechatronics (ICROM), 2015 3rd RSI International Conference on*, 2015, pp. 418–424.

[8] R. C. Arkin, *Behavior-based robotics.* MIT press, 1998.

[9] M. S. Arulampalam, S. Maskell, N. Gordon and T. Clapp, “A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking”, *IEEE Transactions on Signal Processing*, 50, no., pp. 174–188, 2002.

[10] K. J. Åström, “Autonomous control”, in *Future Tendencies in Computer Science, Control and Applied Mathematics: International Conference on the Occasion of the 25th Anniversary of INRIA Paris, France, December 8–11, 1992 Proceedings*, A. Bensoussan and J. P. Verjus, Eds., Berlin, Heidelberg: Springer, 1992, pp. 265–278.

[11] K. J. Åström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers.* Princeton university press, 2010.

[12] F. Bacchus and F. Kabanza, “Using temporal logics to express search control knowledge for planning”, *Artificial Intelligence*, 116, no., pp. 123–191, 2000.

[13] C. Baier, J.-P. Katoen *et al.*, *Principles of model checking.* MIT press Cambridge, 2008, vol. 26202649.

[14] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi and Y. Watanabe, “Modeling and designing heterogeneous systems”, in *Concurrency and Hardware Design: Advances in Petri Nets*, Berlin, Heidelberg: Springer, 2002, pp. 228–273.

[15] G. K. Batchelor, *An Introduction to Fluid Dynamics.* Cambridge University Press, 2000.

[16] T. M. Behrens, K. V. Hindriks and J. Dix, “Towards an environment interface standard for agent platforms”, *Annals of Mathematics and Artificial Intelligence*, 61, no., pp. 261–295, 2010.

[17] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi and M. Hendriks, “UPPAAL 4.0”, in *Third International Conference on the Quantitative Evaluation of Systems - (QEST’06)*, 2006, pp. 125–126.

[18] F. L. Bellifemine, G. Caire and D. Greenwood, *Developing multi-agent systems with JADE.* John Wiley & Sons, 2007, vol. 7.

[19] F. L. Bellifemine, A. Poggi, G. Rimassa and P. Turci, “An object-oriented framework to realize agent systems”, in *WOA*, Parma, Italy, 2000, pp. 52–57.

[20] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins and G. J. Pappas, “Symbolic planning and control of robot motion”, *IEEE Robotics & Automation Magazine*, 14, no., pp. 61–70, 2007.

[21] C. Belta and L. Habets, “Constructing decidable hybrid systems with velocity bounds”, in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, IEEE, vol. 1, 2004, pp. 467–472.

[22] M. Benerecetti, F. Giunchiglia and L. Serafini, “A model checking algorithm for multi-agent systems”, in *Intelligent Agents V: Agents Theories, Architectures, and Languages: 5th International Workshop, ATAL’98 Paris, France, July 4–7, 1998 Proceedings*, J. P. Müller, A. S. Rao and M. P. Singh, Eds., Berlin, Heidelberg: Springer, 1999, pp. 163–176.

[23] M. R. Benjamin, “The interval programming model for multi-objective decision making”, Massachusetts Institute of Technology, Tech. Rep. AIM-2004-021, 2004.

[24] M. R. Benjamin, H. Schmidt, M. P. Newman and J. J. Leonard, *Unmanned Marine Vehicle Autonomy with MOOS-IvP.* Springer, 2012, ch. 2, pp. 1–100.

[25] ——, “Autonomy for unmanned marine vehicles with MOOS-IvP”, in *Marine Robot Autonomy*, L. M. Seto, Ed., New York, NY, USA: Springer, 2013, ch. 2, pp. 47–90.

[26] M. R. Benjamin, H. Schmidt, P. M. Newman and J. J. Leonard, “Nested autonomy for unmanned marine vehicles with MOOS-IvP”, *Journal of Field Robotics*, 27, no., pp. 834–875, 2010.

[27] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri and P. Traverso, “MBP: A model based planner”, in *Proc. of the IJCAI’01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[28] A. Bhatia, L. E. Kavraki and M. Y. Vardi, “Sampling-based motion planning with temporal goals”, in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, IEEE, 2010, pp. 2689–2696.

[29] A. Bhatia, M. R. Maly, L. E. Kavraki and M. Y. Vardi, “Motion planning with complex goals”, *IEEE Robotics & Automation Magazine*, 18, no., pp. 55–64, 2011.

[30] A. Bianco and L. Alfaro, “Model checking of probabilistic and nondeterministic systems”, in *Foundations of Software Technology and Theoretical Computer Science: 15th Conference Bangalore, India, December 18–20, 1995 Proceedings*, P. S. Thiagarajan, Ed., Berlin, Heidelberg: Springer, 1995, pp. 499–513.

[31] J. Bongard, V. Zykov and H. Lipson, “Resilient machines through continuous self-modeling”, *Science*, 314, no., pp. 1118–1121, 2006.

[32] R. H. Bordini, M. Fisher, C. Pardavila, W. Visser and M. Wooldridge, “Model checking multi-agent programs with CASP”, in *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer, 2003, pp. 110–113.

[33] R. H. Bordini, M. Fisher, C. Pardavila and M. Wooldridge, “Model checking AgentSpeak”, in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS ’03, New York, NY, USA: ACM, 2003, pp. 409–416.

[34] R. H. Bordini, M. Fisher, W. Visser and M. Wooldridge, “Verifiable multi-agent programs”, in *Programming Multi-Agent Systems: First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited papers*, M. M. Dastani, J. Dix and A. El Fallah-Seghrouchni, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 72–89.

[35] ——, “Verifying multi-agent programs by model checking”, *Autonomous agents and multi-agent systems*, 12, no., pp. 239–256, 2006.

[36] R. H. Bordini and J. F. Hubner, *Jason, a java-based interpreter for an extended version of agentspeak*, Manual version 0.9.5, 2007.

[37] R. H. Bordini, J. F. Hubner and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason.* Chichester: Wiley, 2007.

[38] E. Bovio, D. Cecchi and F. Baralli, “Autonomous underwater vehicles for scientific and naval operations”, *Annual Reviews in Control*, 30, no., pp. 117–130, 2006.

[39] T. Bradshaw, “How robots are taking over our homes”, *Financial Times*, no., 2015.

[40] M. E. Bratman, *Intention, plans, and practical reason.* Center for the Study of Language and Information, 1987.

[41] M. E. Bratman, D. J. Israel and M. E. Pollack, “Plans and resource-bounded practical reasoning”, *Computational intelligence*, 4, no., pp. 349–355, 1988.

[42] R. A. Brooks, “Elephants don’t play chess”, *Robotics and autonomous systems*, 6, no., pp. 3–15, 1990.

[43] R. E. Bryant, “Graph-based algorithms for boolean function manipulation”, *Computers, IEEE Transactions on*, 100, no., pp. 677–691, 1986.

[44] P. Busetta, N. Howden, R. Rönnquist and A. Hodgson, “Structuring BDI agents in functional clusters”, in *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, Springer, 1999, pp. 277–289.

[45] J. Cannon, K. Rose and W. Ruml, “Real-time heuristic search for motion planning with dynamic obstacles”, *AI Communications*, 27, no., pp. 345–362, 2014.

[46] L. Carloni, M. D. Di Benedetto, A. Pinto and A. Sangiovanni-Vincentelli, “Modeling techniques, programming languages, design toolsets and interchange formats for hybrid systems”, Technical report, IST-2001-38314 WPHS, Columbus Project, Tech. Rep., 2004.

[47] Y. Chen, X. C. Ding, A. Stefanescu and C. Belta, “Formal approach to the deployment of distributed robotic teams”, *IEEE Transactions on Robotics*, 28, no., pp. 158–171, 2012.

[48] P. P. Choi and M. Hebert, “Learning and predicting moving object trajectory: A piecewise trajectory segment approach”, Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-06-42, 2006.

[49] H. M. Choset, *Principles of robot motion: theory, algorithms, and implementation.* MIT press, 2005.

[50] I. Cizelj and C. Belta, “Control of noisy differential-drive vehicles from time-bounded temporal logic specifications”, *The International Journal of Robotics Research*, no., 2014.

[51] E. M. Clarke, E. A. Emerson and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8, no., pp. 244–263, 1986.

[52] E. M. Clarke, O. Grumberg and D. Peled, *Model checking.* MIT press, 1999.

[53] E. M. Clarke, K. L. McMillan, S. Campos and V. Hartonas-Garmhausen, “Symbolic model checking”, in *Computer Aided Verification: 8th International Conference, CAV ’96 New Brunswick, NJ, USA, July 31– August 3, 1996 Proceedings*, R. Alur and T. A. Henzinger, Eds., Berlin, Heidelberg: Springer, 1996, pp. 419–422.

[54] D. C. Conner, A. A. Rizzi and H. Choset, “Composition of local potential functions for global robot control and navigation”, in *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, IEEE, vol. 4, 2003, pp. 3546–3551.

[55] C. Courcoubetis and M. Yannakakis, “Verifying temporal properties of finite-state probabilistic programs”, in *Foundations of Computer Science, 1988., 29th Annual Symposium on*, IEEE, 1988, pp. 338–345.

[56] ——, “The complexity of probabilistic verification”, *Journal of the ACM (JACM)*, 42, no., pp. 857–907, 1995.

[57] S. K. Das and A. A. Reyes, “An approach to integrating HLA federations and genetic algorithms to support automatic design evaluation for multi-agent systems”, *Simulation Practice and Theory*, 9, no., pp. 167–192, 2002.

[58] L. De Alfaro, M. Kwiatkowska, G. Norman, D. Parker and R. Segala, *Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation.* Springer, 2000.

[59] G. De Giacomo, Y. Lespérance and H. J. Levesque, “ConGolog, a concurrent programming language based on the situation calculus”, *Artificial Intelligence*, 121, no., pp. 109–169, 2000.

[60] L. A. Dennis and B. Farwer, “Gwendolen: A BDI language for verifiable agents”, in *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning*, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 2008, pp. 16–23.

[61] L. A. Dennis, B. Farwer, R. H. Bordini and M. Fisher, “A flexible framework for verifying agent programs”, in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, ser. AAMAS ’08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2008, pp. 1303–1306.

[62] L. A. Dennis and M. Fisher, “Programming verifiable heterogeneous agent systems”, in *Programming Multi-Agent Systems: 6th International Workshop, ProMAS 2008, Estoril, Portugal, May 13, 2008. Revised Invited and Selected Papers*, K. V. Hindriks, A. Pokahr and S. Sardina, Eds., Berlin, Heidelberg: Springer, 2008, pp. 40–55.

[63] L. A. Dennis, M. Fisher and M. Webster, “Two-stage agent program verification”, *Journal of Logic and Computation*, no., exv003, 2015.

[64] L. A. Dennis, M. Fisher, M. P. Webster and R. H. Bordini, “Model checking agent programming languages”, *Automated software engineering*, 19, no., pp. 5–63, 2012.

[65] A. Diamond, R. Knight, D. Devereux and O. Holland, “Anthropomimetic robots: Concept, construction and modelling”, *International Journal of Advanced Robotic Systems*, 9, no., 2012.

[66] E. W. Dijkstra, “A note on two problems in connexion with graphs”, *Numerische mathematik*, 1, no., pp. 269–271, 1959.

[67] M. D’Inverno, D. Kinny, M. Luck and M. Wooldridge, “A formal specification of dMARS”, in *Intelligent Agents IV Agent Theories, Architectures, and Languages: 4th International Workshop, ATAL’97 Providence, Rhode Island, USA, July 24–26, 1997 Proceedings*, M. P. Singh, A. Rao and M. J. Wooldridge, Eds., Berlin, Heidelberg: Springer, 1997, pp. 155–176.

[68] G. M.W. M. Dissanayake, P. Newman, H. F. Durrant-Whyte, S. Clark and M. Csobra, “A solution to the Simultaneous Localization And Map building (SLAM) problem”, *IEEE Transactions on Robotics and Automation*, 17, no., pp. 229–241, 2001.

[69] C. Dixon, A. F. Winfield, M. Fisher and C. Zeng, “Towards temporal verification of swarm robotic systems”, *Robotics and Autonomous Systems*, 60, no., pp. 1429–1441, 2012.

[70] R. C. Dorf and R. H. Bishop, *Modern control systems.* Pearson (Addison-Wesley), 1998.

[71] A. Doucet, “On sequential simulation-based methods for bayesian filtering”, Department of Engineering, University of Cambridge, Tech. Rep., 1998.

[72] A. Doucet, N. de Freitas, K. Murphy and S. Russel, “Rao-blackwellised particle filtering for dynamic bayesian networks”, in *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI’00, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 176–183.

[73] H. F. Durrant-Whyte and T. Bailey, “Simultaneous Localization and Mapping: Part i”, *IEEE Robotics & Automation Magazine*, no., 2006.

[74] ——, “Simultaneous Localization and Mapping: Part ii”, *IEEE Robotics & Automation Magazine*, no., 2006.

[75] G. E. Fainekos, H. Kress-Gazit and G. J. Pappas, “Temporal logic motion planning for mobile robots”, in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, 2005, pp. 2020–2025.

[76] G. E. Fainekos, A. Girard, H. Kress-Gazit and G. J. Pappas, “Temporal logic motion planning for dynamic robots”, *Automatica*, 45, no., pp. 343–352, 2009.

[77] E. A. Feinberg and A. Shwartz, *Handbook of Markov Decision Processes: methods and applications.* Springer, 2012, vol. 40.

[78] D. Ferguson and A. Stentz, "Field D*: An interpolation-based path planner and replanner", in *Robotics Research: Results of the 12th International Symposium ISRR*, S. Thrun, R. Brooks and H. Durrant-Whyte, Eds., Berlin, Heidelberg: Springer, 2007, pp. 239–253.

[79] A. Ferrein and G. Lakemeyer, "Logic-based robot control in highly dynamic domains", *Robotics and Autonomous Systems*, 56, no., pp. 980–991, 2008.

[80] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial intelligence*, 2, no., pp. 189–208, 1971.

[81] M. Fisher, "A survey of concurrent METATEM - the language and its applications", in *Temporal Logic: First International Conference, ICTL'94 Bonn, Germany, July 11–14, 1994 Proceedings*, D. M. Gabbay and H. J. Ohlbach, Eds., Berlin, Heidelberg: Springer, 1994, pp. 480–505.

[82] V. Forejt, M. Kwiatkowska, G. Norman and D. Parker, "Automated verification techniques for probabilistic systems", in *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, M. Bernardo and V. Issarny, Eds., Berlin, Heidelberg: Springer, 2011, pp. 53–113.

[83] T. I. Fossen, *Guidance and control of ocean vehicles*. Wiley, 1994.

[84] ——, *Marine Control Systems: Guidance, Navigation and Control of Ships, Rigs and Underwater Vehicles*. Marine Cybernetics, 2002.

[85] R. J. C. Fraser, C. J. Harris, L. W. Mathias and N. J. W. Rayner, "Implementing task-level mission management for intelligent autonomous vehicles", *Engineering Applications of Artificial Intelligence*, 4, no., pp. 257–268, 1991.

[86] E. Frazzoli, M. A. Dahleh and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries", *IEEE Transactions on Robotics*, 21, no., pp. 1077–1091, 2005.

[87] M. Fujita, P. C. McGeer and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation", *Formal methods in system design*, 10, no., pp. 149–169, 1997.

[88] E. Gat, "On three-layer architectures", *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, no., pp. 195–210, 1998, MIT Press.

[89] *Gazebo*, [Accessed: 26th March 2017]. [Online]. Available: `http://gazebosim.org`.

[90] M. P. Georgeff and F. F. Ingrand, "Decision-making in an embedded reasoning system", Australian Artificial Intelligence Institute, Tech. Rep. 479, 1989.

[91] M. P. Georgeff and A. L. Lansky, “Procedural knowledge”, *Proceedings of the IEEE*, 74, no., pp. 1383–1398, 1986.

[92] ——, “Reactive reasoning and planning.”, in *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2*, ser. AAAI’87, vol. 2, AAAI Press, 1987, pp. 677–682.

[93] M. P. Georgeff, B. Pell, M. Pollack, M. Tambe and M. Wooldridge, “The belief-desire-intention model of agency”, in *Intelligent Agents V: Agents Theories, Architectures, and Languages: 5th International Workshop, ATAL’98 Paris, France, July 4–7, 1998 Proceedings*, J. P. Müller, A. S. Rao and M. P. Singh, Eds., Berlin, Heidelberg: Springer, 1998, pp. 1–10.

[94] A. E. Gerevini, P. Haslum, D. Long, A. Saetti and Y. Dimopoulos, “Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners”, *Artificial Intelligence*, 173, no., pp. 619–668, 2009.

[95] J. E. Goodman, J. O’Rourke and K. H. Rosen, *Handbook of discrete and computational geometry.* cRc Press LLc, 2000.

[96] D. L. Hall and J. Llinas, “An introduction to multisensor data fusion”, *Proceedings of the IEEE*, 85, no., pp. 6–23, 1997.

[97] M. J. Hamilton, S. Kemna and D. Hughes, “Antisubmarine warfare applications for autonomous underwater vehicles: The GLINT09 sea trial results”, *Journal of Field Robotics*, 27, no., pp. 890–902, 2010.

[98] T. Han, J.-P. Katoen and D. Berteun, “Counterexample generation in probabilistic model checking”, *Software Engineering, IEEE Transactions on*, 35, no., pp. 241–257, 2009.

[99] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability”, *Formal aspects of computing*, 6, no., pp. 512–535, 1994.

[100] P. E. Hart, N. J. Nilsson and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths”, *IEEE Transactions on Systems Science and Cybernetics*, 4, no., pp. 100–107, 1968.

[101] K. Havelund, M. Lowry and J. Penix, “Formal analysis of a space-craft controller using SPIN”, *Software Engineering, IEEE Transactions on*, 27, no., pp. 749–765, 2001.

[102] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle, “A markov chain model checker”, in *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25 – April 2, 2000 Proceedings*, S. Graf and M. Schwartzbach, Eds., Springer, 2000, pp. 347–362.

[103] K. V. Hindriks, F. S. Boer, W. Hoek and J.-J. C. Meyer, “Formal semantics for an abstract agent programming language”, in *Intelligent Agents IV Agent Theories, Architectures, and Languages: 4th International Workshop, ATAL’97 Providence, Rhode Island, USA, July 24–26, 1997 Proceedings*, M. P. Singh, A. Rao and M. J. Wooldridge, Eds., Berlin, Heidelberg: Springer, 1998, pp. 215–229.

[104] K. V. Hindriks, F. S. Boer, W. v. d. Hoek and J.-J. C. Meyer, “Control structures of rule-based agent languages”, in *Intelligent Agents V: Agents Theories, Architectures, and Languages: 5th International Workshop, ATAL’98 Paris, France, July 4–7, 1998 Proceedings*, J. P. Müller, A. S. Rao and M. P. Singh, Eds., Berlin, Heidelberg: Springer, 1999, pp. 381–396.

[105] K. V. Hindriks, F. S. de Boer, W. Van Der Hoek and J.-J. C. Meyer, “A formal embedding of AgentSpeak (L) in 3APL”, *Advanced Topics in Artificial Intelligence: 11th Australian Joint Conference on Artificial Intelligence, AI’98 Brisbane, Australia, July 13–17, 1998 Selected Papers*, no., pp. 155–166, 1998.

[106] K. V. Hindriks, F. S. De Boer, W. Van der Hoek and J.-J. C. Meyer, “Agent programming in 3APL”, *Autonomous Agents and Multi-Agent Systems*, 2, no., pp. 357–401, 1999.

[107] A. Hinton, M. Kwiatkowska, G. Norman and D. Parker, “PRISM: A tool for automatic verification of probabilistic systems”, in *Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006. Proceedings*, H. Hermanns and J. Palsberg, Eds., Berlin, Heidelberg: Springer, 2006, pp. 441–444.

[108] C. A. R. Hoare, “An axiomatic basis for computer programming”, *Commun. ACM*, 12, no., pp. 576–580, Oct. 1969.

[109] R. Hoffmann, M. Ireland, A. Miller, N. Gethin and S. M. Veres, “Autonomous agent behaviour modelled in PRISM – A case study”, arXiv:1602.00646v2 [cs.SY], 2016.

[110] J. H. Holland, “Complex adaptive systems”, *Daedalus*, no., pp. 17–30, 1992.

[111] G. J. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[112] ——, “The model checker SPIN”, *IEEE Transactions on software engineering*, 23, no., p. 279, 1997.

[113] ——, *The SPIN model checker: Primer and reference manual.* Addison-Wesley Reading, 2004, vol. 1003.

[114] N. Howden, R. Rönnquist, A. Hodgson and A. Lucas, "JACK intelligent agents-summary of an agent infrastructure", in *5th International conference on autonomous agents*, 2001.

[115] M. J. Huber, "JAM: A BDI-theoretic mobile agent architecture", in *Proceedings of the 3rd annual conference on Autonomous Agents*, ser. AGENTS '99, New York, NY, USA: ACM, 1999, pp. 236–243.

[116] G. E. Hughes and M. J. Cresswell, *A new introduction to modal logic.* Psychology Press, 1996.

[117] J. Hunter, F. Raimondi, N. Rungta and R. Stocker, "A synergistic and extensible framework for multi-agent system verification", in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, ser. AAMAS '13, Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 869–876.

[118] P. Izzo, H. Qu and S. M. Veres, "A stochastically verifiable autonomous control architecture with reasoning", in *Decision and Control, 2016. CDC. 55th IEEE Conference on*, IEEE, 2016.

[119] P. Izzo and S. M. Veres, "Intelligent planning with performance assessment for autonomous surface vehicles", in *OCEANS 2015 - Genova*, 2015, pp. 1–6.

[120] R. M. Jensen and M. M. Veloso, "OBDD-based universal planning for synchronized agents in non-deterministic domains", *Journal of Artificial Intelligence Research*, 13, no., pp. 189–226, 2000.

[121] M. Kallmann, "Shortest paths with arbitrary clearance from navigation mashes", in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '10, Aire-la-Ville, Switzerland: Eurographics Association, 2010, pp. 159–168.

[122] ——, "Dynamic and robust local clearance triangulations", *ACM Transactions on Graphics (TOG)*, 33, no., 2014.

[123] M. Kallmann, H. Bieri and D. Thalmann, "Fully dynamic constrained delaunay triangulations", *Geometric Modeling for Scientific Visualization*, no., pp. 241–257, 2003.

[124] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning", *The International Journal of Robotics Research*, 30, no., pp. 846–894, 2011.

[125] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli and S. Teller, "Anytime motion planning using the rrt", in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, 2011, pp. 1478–1483.

[126] L. E. Kavraki, P. Švestka, J.-C. Latombe and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces", *IEEE Transactions on Robotics and Automation*, 12, no., pp. 566–580, 1996.

[127] M. Kloetzer and C. Belta, “Temporal logic planning and control of robotic swarms by hierarchical abstractions”, *IEEE Transactions on Robotics*, 23, no., pp. 320–330, 2007.

[128] S. Koenig and X. Sun, “Comparing real-time and incremental heuristic search for real-time situated agents”, *Autonomous Agents and Multi-Agent Systems*, 18, no., pp. 313–341, 2009.

[129] S. Konur, C. Dixon and M. Fisher, “Analysing robot swarm behaviour via probabilistic model checking”, *Robotics and Autonomous Systems*, 60, no., pp. 199–213, 2012.

[130] H. Kress-Gazit, G. E. Fainekos and G. J. Pappas, “Temporal-logic-based reactive mission and motion planning”, *IEEE transactions on robotics*, 25, no., pp. 1370–1381, 2009.

[131] H. Kress-Gazit, T. Wongpiromsarn and U. Topcu, “Correct, reactive, high-level robot control”, *IEEE Robotics Automation Magazine*, 18, no., pp. 65–74, 2011.

[132] F. Kröger and S. Merz, *Temporal Logic and State Systems*, W. Brauer, J. Hromkovič, G. Rozenberg and A. Salomaa, Eds. Springer, 2008.

[133] A. Kushleyev and M. Likhachev, “Time-bounded lattice for efficient planning in dynamic environments”, in *Robotics and Automation, 2009. ICRA’09. IEEE International Conference on*, IEEE, 2009, pp. 1662–1668.

[134] M. Kwiatkowska, G. Norman and D. Parker, “Stochastic model checking”, in *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, M. Bernardo and J. Hillston, Eds., Berlin, Heidelberg: Springer, 2007, pp. 220–270.

[135] ——, “PRISM 4.0: Verification of probabilistic real-time systems”, in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, G. Gopalakrishnan and S. Qadeer, Eds., Berlin, Heidelberg: Springer, 2011, pp. 585–591.

[136] M. Kwiatkowska and D. Parker, “Advances in probabilistic model checking”, in *Software Safety and Security - Tools for Analysis and Verification*, T. Nipkow, O. Grumberg and B. Hauptmann, Eds., ser. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 33, IOS Press, 2012, pp. 126–151.

[137] M. Lahijanian, S. B. Andersson and C. Belta, “Temporal logic motion planning and control with probabilistic satisfaction guarantees”, *IEEE Transactions on Robotics*, 28, no., pp. 396–409, 2012.

[138] J. C. Latombe, *Robot Motion Planning.* Norwell, MA, USA: Kluwer Academic Publishers, 1991.

[139] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning”, Computer Science Dept., Iowa State University, Technical Report 98-11, 1998.

[140] ——, *Planning Algorithms.* Cambridge University Press, 2006.

[141] ——, “Motion planning: The essentials”, *IEEE Robotics & Automation Magazine*, 18, no., pp. 79–89, 2011.

[142] ——, “Motion planning: Wild frontiers”, *IEEE Robotics & Automation Magazine*, 18, no., pp. 108–118, 2011.

[143] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter and R. B. Scherl, “Foundations of a logical approach to agent programming”, in *Intelligent Agents II Agent Theories, Architectures, and Languages: IJCAI’95 Workshop (ATAL) Montréal, Canada, August 19–20, 1995 Proceedings*, M. Wooldridge, J. P. Müller and M. Tambe, Eds., Berlin, Heidelberg: Springer, 1995, pp. 331–346.

[144] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin and R. B. Scherl, “GOLOG: A logic programming language for dynamic domains”, *The Journal of Logic Programming*, 31, no., pp. 59–83, 1997.

[145] M. Likhachev and A. Stentz, “R* search”, *Lab Papers (GRASP)*, no., p. 23, 2008.

[146] N. K. Lincoln and S. M. Veres, “Natural language programming of complex robotic BDI agents”, *Intelligent and Robotic Systems*, 71, no., pp. 211–230, 2013.

[147] N. K. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher and A. Lisitsa, “Autonomous asteroid exploration by rational agents”, *Computational Intelligence Magazine, IEEE*, 8, no., pp. 25–38, 2013.

[148] J. Liu, N. Ozay, U. Topcu and M. R. Murray, “Synthesis of reactive switching protocols from temporal logic specifications”, *IEEE Transactions on Automatic Control*, 58, no., pp. 1771–1785, 2013.

[149] A. Lomuscio, H. Qu and F. Raimondi, “MCMAS: A model checker for the verification of multi-agent systems”, in *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, A. Bouajjani and O. Maler, Eds., Berlin, Heidelberg: Springer, 2009, pp. 682–688.

[150] T. Lozano-Perez, “Spatial planning: A configuration space approach”, *IEEE transactions on computers*, 100, no., pp. 108–120, 1983.

[151] B. D. Luders, S. Karaman and J. P. How, “Robust sampling-based motion planning with asymptotic optimality guarantees”, in *AIAA Guidance, Navigation, and Control Conference (GNC), Boston, MA*, 2013.

[152] P. Maes, “The agent network architecture (ANA)”, *Acm sigart bulletin*, 2, no., pp. 115–120, 1991.

[153] C. E. Maniere and R. Simmons, “Architecture, the backbone of robotic systems”, in *Robotics and Automation, 2000. Proceedings. ICRA ’00. IEEE International Conference on*, IEEE, vol. 1, 2000, pp. 67–72.

[154] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety.* Berlin: Springer, 1995.

[155] *Marine systems simulator (MSS)*, [Accessed: 26th March 2017]. [Online]. Available: `http://www.marinecontrol.org`.

[156] H. G. Marques and O. Holland, “Architectures for functional imagination”, *Neurocomputing*, 72, no., pp. 743–759, 2009.

[157] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, “PDDL - the planning domain definition language”, no., 1998.

[158] J. McMahon and E. Plaku, “Sampling-based tree search with discrete abstractions for motion planning with dynamics and temporal logic”, in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2014, pp. 3726–3733.

[159] ——, “Robot motion planning with task specifications via regular languages”, *Robotica*, no., pp. 1–24, 2015.

[160] K. L. McMillan, *Symbolic Model Checking.* Boston, MA: Springer, 1993.

[161] A. M. Meystel and J. S. Albus, *Intelligent Systems: architecture, design and control.* Wiley, 2002.

[162] M. Montemerlo, S. Thrun, D. Koller and B. Wegbreit, “FastSLAM: A factored solution to the simultaneous localization and mapping problem”, in *Proceedings of AAAI National Conference on Artificial Intelligence*, AAAI Press, 2002, pp. 593–598.

[163] J. P. Müller, M. Pischel and M. Thiel, “Modeling reactive behaviour in vertically layered agent architectures”, *Intelligent Agents*, 890, no., pp. 261–276, 1995.

[164] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu *et al.*, “CLARAty: Challenges and steps toward reusable robotic software”, *International Journal of Advanced Robotic Systems*, 3, no., pp. 23–30, 2006.

[165] M. P. Newman, “MOOS - a mission oriented operating suite”, Massachusetts Institute of Technology, Tech. Rep. OE2003-07, 2003.

[166] K. W. Ng and C.-K. Luk, “I+: A multiparadigm language for object-oriented declarative programming”, *Computer Languages*, 21, no., pp. 81–100, 1995.

[167] E. Norling and F. E. Ritter, “Embodying the JACK agent architecture”, in *AI 2001: Advances in Artificial Intelligence: 14th Australian Joint Conference on Artificial Intelligence Adelaide, Australia, December 10–14, 2001 Proceedings*, M. Stumptner, D. Corbett and M. Brooks, Eds., Berlin, Heidelberg: Springer, 2001, pp. 368–377.

[168] J. R. Norris, *Markov chains.* Cambridge university press, 1998.

[169] E. P. Pednault, “ADL and the state-transition model of action”, *Journal of logic and computation*, 4, no., pp. 467–512, 1994.

[170] A. Pinto, L. P. Carloni, R. Passerone and A. Sangiovanni-Vincentelli, “Interchange semantics for hybrid system models”, in *Procedings of the 5th International Conference on Mathematical Modeling*, 2006, pp. 1–9.

[171] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni and R. Passerone, “Interchange formats for hybrid systems: Review and proposal”, in *Hybrid Systems: Computation and Control: 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005. Proceedings*, M. Morari and L. Thiele, Eds., Berlin, Heidelberg: Springer, 2005, pp. 526–541.

[172] N. Piterman, A. Pnueli and Y. Sa’ar, “Synthesis of reactive(1) designs”, in *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings*, E. A. Emerson and K. S. Namjoshi, Eds., Berlin, Heidelberg: Springer, 2006, pp. 364–380.

[173] E. Plaku and S. Karaman, “Motion planning with temporal-logic specifications: Progress and challenges”, *AI Communications*, 29, no., pp. 151–162, 2015.

[174] E. Plaku, L. E. Kavraki and M. Y. Vardi, “Motion planning with dynamics by a synergistic combination of layers of planning”, *IEEE Transactions on Robotics*, 26, no., pp. 469–482, 2010.

[175] E. Plaku and J. McMahon, “Motion planning and decision making for underwater vehicles operating in constrained environments in the littoral”, in *Towards Autonomous Robotic Systems: 14th Annual Conference, TAROS 2013, Oxford, UK, August 28–30, 2013, Revised Selected Papers*, A. Natraj, S. Cameron, C. Melhuish and M. Witkowski, Eds., Berlin, Heidelberg: Springer, 2014, pp. 328–339.

[176] G. D. Plotkin, “A structural approach to operational semantics”, Aarhus University, Computer Science Dept., Tech. Rep., 1981.

[177] A. Pnueli, “The temporal logic of programs”, in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, IEEE, 1977, pp. 46–57.

[178] M. E. Pollack, “Plans as complex mental attitudes”, *Intentions in communication*, no., pp. 77–103, 1990.

[179] ——, “The uses of plans”, *Artificial Intelligence*, 57, no., pp. 43–68, 1992.

[180] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng, “ROS: An open-source Robot Operating System”, in *ICRA workshop on open source software*, vol. 3, 2009, p. 5.

[181] A. S. Rao, “AgentSpeak (L): BDI agents speak out in a logical computable language”, in *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW ’96 Eindhoven, The Netherlands, January 22–25, 1996 Proceedings*, Berlin, Heidelberg: Springer, 1996, pp. 42–55.

[182] ——, “Decision procedures for prepositional linear-time belief-desire-intention logics”, in *Intelligent Agents II Agent Theories, Architectures, and Languages: IJCAI’95 Workshop (ATAL) Montréal, Canada, August 19–20, 1995 Proceedings*, M. Wooldridge, J. P. Müller and M. Tambe, Eds., Berlin, Heidelberg: Springer, 1996, pp. 33–48, ISBN: 978-3-540-49594-9. DOI: `10.1007/3540608052_57`.

[183] A. S. Rao and M. P. Georgeff, “A model-theoretic approach to the verification of situated reasoning systems”, in *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, ser. IJCAI’93, Chambery, France: Morgan Kaufmann Publishers Inc., 1993, pp. 318–324.

[184] A. S. Rao, M. P. Georgeff *et al.*, “BDI agents: From theory to practice.”, in *Proceedings of the 1st International Conference on Multiagent Systems*, vol. 95, 1995, pp. 312–319.

[185] J. H. Reif, “Complexity of the generalized mover’s problem.”, DTIC Document, Tech. Rep., 1985.

[186] P. Ridao, J. Batlle and M. Carreras, “O^2CA^2, a new object oriented control architecture for autonomy: The reactive layer”, *Control Engineering Practice*, 10, no., pp. 857–873, 2002.

[187] J. Rosenblatt, S. Williams and H. Durrant-Whyte, “A behavior-based architecture for autonomous underwater exploration”, *Information Sciences*, 145, no., pp. 69–87, 2002.

[188] S. J. Rosenschein and L. P. Kaelbling, “A situated view of representation and control”, *Artificial Intelligence*, 72, no., 1995.

[189] S. Russell, P. Norvig and A. Intelligence, *A modern approach.* Citeseer, 1995, vol. 25, p. 27.

[190] J. J.M. M. Rutten, M. Kwiatkowska, G. Norman and D. Parker, *Mathematical techniques for analyzing concurrent and probabilistic systems.* American Mathematical Soc., 2004.

[191] J. T. Schwartz, *Planning, geometry, and complexity of robot motion.* Intellect Books, 1987.

[192] J. T. Schwartz and M. Sharir, "On the "piano movers' " problem I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers", *Communications on pure and applied mathematics*, 36, no., pp. 345–398, 1983.

[193] Y. Shoham, "Agent-oriented programming", *Artificial intelligence*, 60, no., pp. 51–92, 1993.

[194] B. Siciliano and O. Khatib, *Springer handbook of robotics.* Springer, 2008.

[195] B. I. Silva, O. Stursberg, B. H. Krogh and S. Engell, "An assessment of the current status of algorithmic approaches to the verification of hybrid systems", in *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, IEEE, vol. 3, 2001, pp. 2867–2874.

[196] R. Simmons and D. Apfelbaum, "A task description language for robot control", in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, IEEE, vol. 3, 1998, pp. 1931–1937.

[197] A. Stentz, "Optimal and efficient path planning for partially-known environments", in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, IEEE, 1994, pp. 3310–3317.

[198] R. Sutton and G. N. Roberts, *Advances in Unmanned Marine Vehicles.* IET, 2006.

[199] S. Thrun, W. Burgard and D. Fox, *Probabilistic Robotics.* The MIT Press, 2005.

[200] U. Topcu, N. Ozay, J. Liu and M. R. Murray, "On synthesizing robust discrete controllers under modeling uncertainty", in *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, ser. HSCC '12, New York, NY, USA: ACM, 2012, pp. 85–94.

[201] A. Ulusoy and C. Belta, "Receding horizon temporal logic control in dynamic environments", *The International Journal of Robotics Research*, 33, no., pp. 1593–1607, 2014.

[202] *GOAL*, [Accessed: 26th March 2017]. [Online]. Available: `http://ii.tudelft.nl/trac/goal`.

[203] *MOOS*, [Accessed: 26th March 2017]. [Online]. Available: `http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php`.

[204] *MOOS-IvP*, [Accessed: 26th March 2017]. [Online]. Available: `http://www.moos-ivp.org`.

[205] *PRISM - probabilistic symbolic model checker*, [Accessed: 26th March 2017]. [Online]. Available: `http://www.prismmodelchecker.org/`.

[206] *ROS - robotics operating system*, [Accessed: 26th March 2017]. [Online]. Available: `http://www.ros.org`.

[207] A. J. Van Der Schaft and J. M. Schumacher, *An introduction to hybrid dynamical systems.* Springer London, 2000, vol. 251.

[208] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite state programs", in *Foundations of Computer Science, 1985., 26th Annual Symposium on*, IEEE, 1985, pp. 327–338.

[209] C. I. Vasile and C. Belta, "Sampling-based temporal logic path planning", in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2013, pp. 4817–4822.

[210] S. M. Veres, "Principles, architectures and trends in autonomous control", in *Autonomous Agents in Control, 2005. The IEE Seminar on (Ref. No. 2005/10986)*, 2005, pp. 1–9.

[211] ——, *Natural Language Programming of Agents and Robotic Devices.* London: Sysbrain Ltd, 2008.

[212] ——, "Mission capable autonomous control systems in the oceans, in the air and in space", in *Brain-Inspired Information Technology*, A. Hanazawa, T. Miki and K. Horio, Eds., Berlin, Heidelberg: Springer, 2010, pp. 1–10.

[213] S. M. Veres, L. Molnar, N. K. Lincoln and C. Morice, "Autonomous vehicle control systems - a review of decision making", *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 225, no., pp. 155–195, 2011.

[214] S. M. Veres, L. Molnar and N. K. Lincoln, *The Cognitive Agents Toolbox (CAT) - programming autonomous vehicles*, SysBrain Ltd, Southampton, 2009.

[215] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, "Model checking programs", *Automated Software Engineering*, 10, no., pp. 203–232, 2003.

[216] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras and H. Das, "The CLARAty architecture for robotic autonomy", in *Aerospace Conference, 2001, IEEE Proceedings.*, IEEE, vol. 1, 2001, pp. 121–132.

[217] E. Waltz, J. Llinas *et al.*, *Multisensor data fusion.* Artech house Norwood, MA, 1990, vol. 685.

[218] D. J. Webb and J. van den Berg, "Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics", in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, IEEE, 2013, pp. 5054–5061.

[219] P. Wegner, "Concepts and paradigms of object-oriented programming", *ACM SIGPLAN OOPS Messenger*, 1, no., pp. 7–87, 1990.

[220] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence.* MIT press, 1999.

[221] E. M. Wolff, U. Topcu and R. M. Murray, “Optimal control with weighted average costs and temporal logic specifications”, *Proc. of Robotics: Science and Systems VIII*, no., 2012.

[222] M. J. Wooldridge, “The logical modelling of computational multi-agent systems”, PhD thesis, Citeseer, 1992.

[223] ——, *Reasoning about rational agents.* MIT press, 2000.

[224] ——, *An Introduction to MultiAgent Systems.* Chichester: Wiley, 2002.

[225] M. J. Wooldridge and N. R. Jennings, “Intelligent agents: Theory and practice”, *The Knowledge Engineering Review*, 10, no., pp. 115–152, 1995.

[226] G. Yasuda, “An object-oriented network environment for computer vision based multirobot system architectures”, in *Proceedings of the 20th International Conference on Computers and Industrial Engineering*, 1996, p. 1199.

[227] ——, “An object-oriented multitasking control environment for multirobot system programming and execution with 3d graphic simulation”, *International journal of production economics*, 60, no., pp. 241–250, 1999.