Science of Computer Programming 139 (2017) 36–55

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A Python framework for programming autonomous robots using a declarative approach

Loris Fichera [1], Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro *

Department of Mathematics and Computer Science, University of Catania, Italy

ARTICLE INFO

Article history:
Received 23 February 2016
Received in revised form 26 January 2017
Accepted 30 January 2017
Available online 9 February 2017

Keywords:
Robot programming
BDI model
AgentSpeak(L)
Python
Operator overloading

ABSTRACT

This paper describes PROFETA (standing for Python RObotic Framework for dEsigning sTrAtegies), a framework for the programming of autonomous robots based on the Belief-Desire-Intention (BDI) software model. PROFETA is inspired by AgentSpeak(L), a formal language for the creation of BDI software agents. The framework is implemented in Python, and utilizes the metaprogramming capabilities offered by this language to implement the operational semantics of AgentSpeak(L). PROFETA provides a flexible environment offering both traditional *object-oriented imperative constructs* and *declarative constructs*, enabling the definition of a robot's high-level behavior in a simple, natural way. The contributions of this paper, in the area of software design and development, are: *(i)* a methodology, equipped with suitable technical solutions, to extend the Python programming language with AgentSpeak(L) declarative constructs; and *(ii)* a unified environment enabling software components for robots to be developed using a *single* language (Python) within a *single* runtime environment (the Python virtual machine). A comparison between PROFETA and other similar frameworks is provided, illustrating common aspects and key differences.

1. Introduction

The operation of an autonomous robotic system necessitates a flexible software architecture, capable of integrating the many components required for the robot control and coordination. These typically include a number of heterogeneous pieces of software, ranging from low-level motor control up to automated action planning and execution. A natural approach is to organize software components in a hierarchy of layers [1–5], depending on their level of abstraction from the hardware. This allows designers to decouple the code that implements "low-level" tasks, which are typically related to control and sensing, from "middle-level" (e.g. coordination of multiple actuators) and "high-level" activities (e.g. reasoning and path planning). Layered architectures impose a clear separation between the different software parts, thus improving maintainability of the codebase, and facilitate the addition of novel features by providing clear programming interfaces between layers.

In layered architectures, the top layer traditionally controls the behavior of the robot, as it relates to its goals and its perception of the environment. In many simple applications, the behavior is defined by a finite-state machine (FSM) [6–9]. This is a convenient approach and is relatively simple to implement, but does not always scale up straightforwardly: complex behaviors, dealing with the need to adapt to continuous changes in the environment, require large FSMs that are difficult to design and maintain.

* Corresponding author.
E-mail addresses: messina@dmi.unict.it (F. Messina), santoro@dmi.unict.it (C. Santoro).
[1] Now with the Department of Mechanical Engineering, Vanderbilt University, Nashville, TN 37235 USA.

http://dx.doi.org/10.1016/j.scico.2017.01.003
0167-6423/

More sophisticated approaches to programming behavior offer explicit support for the implementation of reasoning processes, rather than the simple execution of condition–action mappings (as in the case of FSMs). Such approaches have been developed in the area of *intelligent agents* [10,11], and in particular of *agent-oriented programming* (AOP) [12,13]. Among these, an interesting paradigm is the Belief–Desire–Intention (BDI) model [14], which is inspired by Bratman's theory of human practical reasoning [15]. Briefly, BDI is based on the concept that an agent has certain goals (*desires*) and a set of *plans* to achieve them; plans are selected, thus becoming *intentions*, depending on the agent's perception of the circumstances (represented by a set of *beliefs*). Beliefs, desires and intentions are specified at a high level, often using a powerful logic/declarative approach: this enables complex behaviors to be implemented, still keeping code transparent and readable.

Integration of a BDI-based layer in a robotic software architecture often results in a mixture of programming languages, styles, and runtime environments: low-level tasks are generally implemented in an imperative language (e.g. C [16] or assembly [17]), while, in many cases, BDI systems utilize declarative languages with a syntax and semantics inspired by either Prolog or LISP [18–22] (even if there are also solutions that exploit an imperative approach). From a software engineering standpoint, developing and maintaining such an architecture would require the coordination of multiple development teams, with specialists for each of the programming languages involved, leading to higher human resources costs. More generally, the proliferation of different programming languages within the same architecture results in software systems that are potentially hard to control, debug and maintain [23–26].

A way to tackle the complexity related with multiple development languages/environments is to adopt an agent-oriented software engineering (AOSE) methodology [27–30], equipped with a CASE tool. This provides powerful assistance in conceptualizing the system with a design process so made essentially visual, and at the same time carries out automatic code generation, thus masking development language diversity and related problems. Yet, this approach, albeit useful, is far from ideal, and should not be seen as the "silver bullet" of robotic systems design. Indeed, it is not uncommon for the generated code to be hard to read and maintain, which also hampers the debugging process. We also observe that this is essentially an agent-oriented approach, typically focusing on the multi-agent interaction aspect, hardly useful for robotic applications.

In this work, we tackle the noted issues, proposing as a solution a novel, unified programming environment that smoothly and elegantly integrates imperative and declarative constructs, thereby enabling developers to rapidly create and combine the different parts of a robotic software architecture. Specifically, we propose a framework, called PROFETA (*Python RObotic Framework for dEsigning sTrAtegies*), which builds on the Python programming language and extends it to implement the operational semantics required to define and run BDI agents. PROFETA runs on the Python virtual machine, and extends Python with a set of declarative constructs inspired by AgentSpeak(L), a well-known BDI kernel language [14,15]. Such constructs were defined through Python's operator overloading, and enable the description of a robot's behavior, while Python's object-oriented/imperative constructs can be used for the implementation of middle and low-level tasks. This paper describes the current state of development of PROFETA, presenting its basics, syntax and architecture, and including a case-study illustrating how to design and develop a robotic application within PROFETA.

The outline of the paper is as follows. Section 2 provides an overview of the BDI model. Section 3 describes related approaches and highlights their differences from PROFETA. Section 4 presents the basic PROFETA model, with its syntax and semantics. Section 5 describes PROFETA's software architecture, presenting its basic classes along with the principles exploited to endow Python with logic/declarative capabilities. Section 6 presents a concrete case study, aimed at showing how PROFETA can be used in an industrial scenario. Section 7 evaluates some distinctive aspects of PROFETA, by comparing them with the capabilities provided by other BDI frameworks. Finally, Section 8 presents the authors' conclusions and illustrates future research directions.

2. Overview of the BDI model

This section presents a brief overview of the Belief–Desire–Intention (BDI) model, which constitutes the foundation of PROFETA. For an exhaustive description of the BDI model, the reader is referred to [15,14].

The BDI paradigm is derived from a philosophical theory of human practical reasoning, and assumes that the *mental* state of an intelligent system (an agent or robot[2]) consists of three fundamental *attitudes*: *Beliefs*, *Desires* and *Intentions*.

Beliefs represent the *informational* state of the agent, which includes the information about its own internal state and the knowledge about the external world. An agent's beliefs are subject to changes due to the data received through *sensors*, which monitor the external world, and as a result of a reasoning process, which is capable of inferring new knowledge. Beliefs are stored in a *knowledge base*, which can be queried to check if a specific information is known to the robot, or to determine if the robot/environment is in a certain state.

Desires, or more commonly *Goals*, represent the *motivational* state of the robot, i.e. a set of objectives it wants to achieve; for instance, picking an object, avoiding an obstacle, searching for an item, reaching a certain position, etc. In practice, in order to achieve its objectives as it operates in its environment, a robot can be endowed with several *plans* specifying the sequence of actions to be carried out. A plan becomes an **Intention** after a *deliberation process* that entails the selection of the "most appropriate plan" to achieve a given goal.

[2] Hereafter, we will use the words "robot" and "agent" interchangeably.

A plan representing a goal is selected only if it is *consistent* with the current set of beliefs of the agent. Thus, the knowledge possessed by the agent becomes a precondition in determining the proper goal to be achieved. The execution of a plan may cause the knowledge base to be updated, due to either an explicit revision of current beliefs, or modifications of the environment state detected through the sensors. The new state of the knowledge base can, in turn, trigger the achievement of a new goal, thus making the reasoning process proceed.

3. BDI and other reasoning based approaches to robot development

As of today, numerous tools and methodologies have been developed to define the behavior of an autonomous robot. The BDI model has been mainly introduced to model the behavior of agent-based and multi-agent systems but its principles have also been applied to robotic systems [31–35]. On the other hand, in the context of *agent-based platforms*, the literature abounds with a plethora of proposals [36], but only few of them have been used to actually program robotic systems [37–39, 35,40].

One of the first advanced reasoning approaches is the generic architecture of the Procedural Reasoning Systems (PRS) [41, 42]. Known implementations of PRS are the PRS-CL [43,44] and dMARS (distributed multi-agent reasoning system) [45]. Both platforms use an ad-hoc LISP-like declarative language to express plans, and provide a graphical tool to help the developer in designing the system.

The JACK framework [22] is an evolution of dMARS and is fully Java-based. The main feature of JACK is JAL, the *Jack Agent Language*, an extension of the Java language; it exhibits characteristics of logic languages (like logical variables), and syntactic constructs suitable to define and handle beliefs, events and plans. JAL extends Java syntax by including the definition of agent-oriented constructs and a set of statements that can operate on BDI-specific data structures in order to perform reasoning activities. JACK main entities are represented by classical BDI constructs: *agents* that handle *beliefs* and behave by executing *plans* triggered by *events*; all these entities can be defined as classes featuring usual object-oriented characteristics, like inheritance. An interesting language construct of JACK is the definition of the *capabilities*, entities introduced, for the BDI-model, in [46]; they are used to encapsulate pieces of a—more or less complex—behavior and can include plans, events and beliefs. Capabilities are mainly intended to be used as “pluggable components” that favor software reuse and thus facilitate the design and development of a BDI agent-based application.

JACK also provides multi-agent support, including an agent communication model. Finally, a graphical environment is provided to facilitate design and development in JACK of BDI-based agent systems [47].

In short, JACK extends Java to provide BDI-aware programming constructs. A different approach is taken in *Jason* [21,48, 49], which runs in a Java environment (like JACK), but provides its own ad-hoc language to express the BDI behavior of an agent. Jason is considered the reference implementation of AgentSpeak(L) [14,19], a kernel, logic-based, BDI language which is also the basis for our PROFETA framework. Jason has been mainly employed in multi-agent programming platforms, such as JaCaMo [50], but has also found a few robot programming applications [49,35,51]. Its architecture is based on a clear separation between the behavior implementation and the interface with the environment of the target agent/robot; behavior is specified with ease in the logic/declarative Jason language, while the other parts are programmed in plain Java. Jason comes with a graphical development tool that allows programmers to write and run Jason programs; a powerful debugger is also provided. Since Jason and PROFETA share the same inspiration, i.e., AgentSpeak(L), an in-depth comparison between them is in order and can be found in Section 7.

Another noteworthy BDI Java-based framework is Jadex [52], a BDI reasoner built on top of the JADE middleware [7]. One of the relevant differences between Jadex and other BDI-based agent-oriented frameworks is represented by the introduction of *explicit goals* [53], a notion which is distinct from that of *plan*. Jadex plans, implemented as Java objects, specify what the agent has to do when a certain triggering event occurs in the environment; among plan actions, a designer can specify the achievement of certain goals. Jadex goals are defined with a *type* and a *context precondition*. The type, used to model the different kinds of goals that may occur in real-life reasoning [53], can be one of: *perform*, *achieve*, *query*, and *maintain*. The context precondition is a predicate that must be true for the goal to be activated. Jadex goals can also include the specification of the set of actions to be performed by the agent to achieve that goal, and can contain sub-goals, thus allowing the specification of goal hierarchies. Goals are collected in the *goalbase*, which is accessed by the reasoning component that takes care of selecting and executing goals, on the basis of their *lifecycle* (*option*, *active* or *suspended*) and dependence from other (sub)goals. A further interesting feature of Jadex are *capabilities*, whereby different reasoning elements of a BDI agent (beliefs, goals, plans, events) can be grouped together into a reusable module, addressing a meaningful set of functionalities; this ability (present in JACK too) affords a degree of encapsulation and reusability. The development model adopted by Jadex consists of: *(i)* an XML *Agent Definition File* (ADF) defining the various BDI entities (beliefs, goals and plans), together with their attributes and parameters, and *(ii)* Java code implementing the body of plans (a Java API enables access to BDI entities). While such a solution may seem reasonable, it confronts developers with a problem if an automatic design tool is not available: handling XML files from text editors is clumsy at best, especially because of XML containing Java fragments required to properly interface the two domains. As for the use of Jadex in robotics, apart from the examples provided in the reference papers [52,53], some significant experiments in the field of multi-robotic systems are reported in [33].

In the context of non-BDI approaches, there are several notable proposals aimed at supporting reasoning in robotic environments, like those based on expert systems, put forward [26,54,4] by the authors of this paper, and the goal-based approaches, of which an outstanding example is GOAP (Goal-Oriented Action Planning) [55]. GOAP introduces a decision-

making architecture, specifically designed for games, which supports the definition of actions, goals (conditions an agent wants to satisfy), and plans (sequences of actions intended to reach a desired goal). The developer has to define the pre-conditions to be satisfied in order to execute a specific action, along with its effects on the state of the world, and the cost associated with the action itself. A planner, using an A^*-like search algorithm, selects a sequence of actions that satisfies the desired goal at the "minimum cost". Two projects we are aware of that implement GOAP are "Emotional GOAP" [56] and pyGOAP [57], released by the project PyGame. An attempt at adopting the GOAP paradigm to specify the behavior of autonomous robots is the RGOAP (Robotic Goal Oriented Action Planning) project [58], presented in [59].

In recent years, the authors of this work developed the GOLEM [60] approach, which shares some aspects with GOAP. GOLEM is an abstract framework for autonomous robot programming, where the behavior of a robot is described by mimicking human behavior, organizing activities into goals and sub-goals, linked to one another through specific relationships. As in GOAP, execution of goals is not determined by a prefixed sequence; instead, the next goal to achieve is selected on the basis of an evaluation of its "opportunity". This is a design choice meant to improve the autonomy enjoyed by a GOLEM-based robot.

Behavior trees (BT) [61] are another reasoning approach developed, like GOAP, to model non-players characters for games. A BT is a directed acyclic graph that specifies a computation executed as the left-to-right traversal of its nodes, starting from the root. The root orderly "ticks" each child, thus enabling it. Any enabled non-leaf node will recursively do the same with its offspring. In reply to the enabling tick, a node may return *success*, *failure* or *running*. A non-leaf returns *running* as soon as one of its children reports to be still *running*. A *sequence* non-leaf returns *success* if all its children succeed, or *failure* as soon as a child is found to have failed. A *fallback* or *selector* non-leaf returns *success* as soon as a child succeeds, or *failure* only after having tried all its children and found them failed. Leaf nodes represent action or conditions to be directly evaluated. An interesting extension of BT purports to model *emotions* [62], by means of an additional *emotional* selector node. Recent efforts [63] have been devoted to adapt BT to robotic and control applications, although, as the authors of [63] themselves remark, BT literature lacks the consistency and mathematical rigor required for this field. Thus, they endeavor to provide an accurate definition of BTs, as the conceptual groundwork for a unified BT framework, and then show its applicability to a real robotics scenario. For this purpose, they design, implement and exploit a layered, open-source, BT library for the Robot Operating System (ROS) [64]. Based on their experience, the authors point out two main difficulties pertaining to the recursive nature of the BT execution model: *(i)* BT implementations exploit recursive function calls, which may cause stack overflow for huge trees; moreover, *(ii)* each "execution tick" traversing the BT from the root requires a large number of checks over the state space of the actions in the tree. As a partial solution to *(ii)*, the traversal and the state checking are executed asynchronously (accepting the latter may lag behind).

We end this short survey of reasoning-based approaches to robotic design with some notes on our PROFETA framework. It was first proposed in [65,66]. The first release was a bare implementation of AgentSpeak(L). However, continued use and testing in our robotic platforms have driven the development of PROFETA, causing it to undergo major modifications from both its original release and inspiring language: the execution semantics has been revised making it more simple and flexible, while the language now includes additional belief constructs (*reactors* and *singleton-beliefs*) and the convenient abstraction of *stages*. As for the execution platform, the present version has been enriched with the explicit introduction of *sensors*, as well as the option of performing sensing and actuation activities in an asynchronous way. All of these, and other aspects, are dealt with in the following sections.

4. PROFETA basics, syntax and semantics

This section provides an overview of PROFETA. We first introduce the basic entities, describe the syntax of the language, and finally illustrate how a typical PROFETA program is structured.

4.1. Basic entities

Like other implementations of the BDI paradigm (cf. Section 2), PROFETA involves the following basic entities: *beliefs*, *goals* and *plans*. In particular, PROFETA supports two kinds of plans: **Goal Plans**, describing the necessary steps to achieve a certain *goal*; and **Reactive Plans**, which are executed upon the occurrence of a certain event, e.g. a change in the knowledge base or failure of another plan. A plan is composed of three basic parts:

- The **Head**, which for *goal plans* consists of the goal name and a list of parameters; for *reactive plans*, this is the specification of the triggering event.
- The **Context Condition** (or simply the **Context**), i.e. a first-order logic predicate specifying the beliefs that must be present in the knowledge base in order to allow the execution of the plan; if the condition is false, the plan cannot be selected for execution.
- The **Body**, i.e. the code of the plan, containing a list of actions which are executed in sequence; actions may include changes to the knowledge base, requests to achieve a goal, or specific actions to be performed onto the environment.

In PROFETA, plans can be organized in **stages**, which represent distinct phases of the robot's operation. Stages provide a useful abstraction to group together plans, facilitating code organization and maintenance. Also, they offer a mechanism

prog	::=	$line_1$ ... $line_n$.	($n \geq 1$)
line	::=	*stage* | *plan*	
stage	::=	"`stage(`" *literal* "`)`"	
plan	::=	*r_plan* | *g_plan*	
r_plan	::=	*r_hd* "`>>`" *aseq*	
g_plan	::=	*g_hd* "`>>`" *aseq*	
r_hd	::=	*evt* | *evt* "`/`" *ctx*	
g_hd	::=	*goal* | *goal* "`/`" *ctx*	
evt	::=	"+"*bel* | "−"*bel* | "−"*goal*	
ctx	::=	$cond_1$ "`&`" ... "`&`" $cond_n$	($n \geq 1$)
cond	::=	*bel* | *lambda*	
lambda	::=	`(lambda :` "*any Python boolean expression*" `)`	
aseq	::=	"[" $action_1, \ldots, action_n$ "]"	($n \geq 0$)
action	::=	`A(`$t_1, \ldots, t_n$`)` | "`set_stage(`" *literal* "`)`"	
		| "*any Python statement*"	
		| "+"*bel* | "−"*bel* | *goal* | "−"*goal*	($n \geq 0$)
bel	::=	*at*	
goal	::=	*at*	
at	::=	`P(`$t_1, \ldots, t_n$`)`	($n \geq 0$)
`A`	::=	`atom`	
`P`	::=	`atom`	
t	::=	`term`	
literal	::=	*any valid Python string*	

Fig. 1. PROFETA syntax.

to quickly enable/disable a set of plans: plans that belong to a stage (referred to as **stage plans**) can only be executed if the program is currently in that stage—the run-time sequence of stages can be controlled using the `set_stage` command. Conversely, a **global plan** does not belong to a specific stage and may be triggered at any time.

In addition to beliefs, plans and stages, which encapsulate the behavioral aspects of the agent, PROFETA implements two additional entities: **sensors** and **actions**. The former collect information from the environment and update the agent's knowledge base accordingly, asserting or retracting beliefs. The latter represent operations that the agent can do to change the status of the environment.

4.2. Language syntax

The syntax of PROFETA is illustrated in Fig. 1. The language implements a logical-declarative paradigm, and its semantics is defined through the operator overloading mechanism of Python [67,68]. PROFETA code is therefore made up of valid Python instructions that are ultimately processed by the PROFETA reasoning engine. As an example, the expression `+my_position(0,0)` causes the addition of a belief (`my_position`) to the knowledge base; the operator + was overloaded to produce such a behavior. These aspects will be covered in greater detail in the next section.

Goals and *beliefs* are the fundamental constructs of PROFETA. They are expressed using a Prolog-like syntax, i.e. logic atomic formulas followed by an optional list of parameters. E.g., the belief `my_position(0,0)` seen earlier has two numerical parameters (presumably meant to identify a 2D point in space). In general, goals and beliefs can have multiple parameters of different types (e.g., `obstacle_position(0,"in_front")`), or have no parameter at all (e.g., `object_got()`); any valid Python type is admissible, including strings, *provided they do not begin by an uppercase letter*. Parameters may also be free variables, which may be assigned a value at runtime. Free variables are enclosed within double quotes and their name must begin with an uppercase letter, e.g. `my_position("X","Y")`. Variable assignment is performed through a Prolog-style pattern matching mechanism: for instance, the belief `my_position("X","Y")` can be matched against `my_position(100,200)`, resulting in variables `X` and `Y` being assigned 100 and 200, respectively. PROFETA parameters with no free variables are said to be "grounded".

Beside *goals* and *beliefs*, the basic building blocks of PROFETA code are *actions*, i.e., operations that are executed *atomically* to make the robot "do something" by controlling its actuators. Examples of actions include driving an arm to pick an object, or moving the robot to a new position. Actions have the same syntax of *goals* and *beliefs*; thus, expressions like `move_to(100,200)`, `move_to("X","Y")` or `go_home()` are all valid action representations.

A PROFETA program contains a number of stages and plans. Plans can be defined globally or within a stage. Global plans come first, followed by stages. A stage is declared using the `stage(`*name*`)` command, and comprises all the plans defined after it and before the next stage command, if any.

A plan is defined by specifying: *(i)* what triggers its execution, *(ii)* the context needed for its execution to be enabled, and *(iii)* the list of actions the plan involves.

A goal plan (*g_plan* in Fig. 1) has a name, the goal itself, with parameters. Execution of a goal plan is simply triggered by the pursuit of its goal name. For a reactive plan (*r_plan*), the trigger is the occurrence of a specific event, represented by either the addition/retraction of a belief from the knowledge base ($+bel/-bel$), or the failure of a goal ($-goal$). Such events may be specified with free variables, and become grounded at run-time through the same pattern-matching mechanism described earlier; as an example, consider the expression `+my_position(100,"Y")`, denoting a triggering event occurring when the belief `my_position` is asserted with the first parameter equal to 100 (and any second parameter): should this

```
1  pick_new_pallet() >> [ set_stage("area-scan") ]
2
3  stage("area-scan")
4  +start() >> [ drive_to("start") ]
5  #...
6  +pallet("X","Y") >> [ stop_robot(), set_stage("pick") ]
7
8  stage("pick")
9  +start() / pallet("X", "Y") >> [ rotate_to(90), forward_slow("X"), activate_bumpers() ]
```

Fig. 2. Some example plans.

occur at run-time, for any value of `my_position()`'s second parameter, the plan will be executed, and the free variable `"Y"` will become bound to that value, which will thereby be available, as `"Y"`, within the scope of the plan.

The second (optional) component of a plan is the *context condition* (*ctx* in Fig. 1). This is a condition, on the state of the knowledge base, which must be satisfied for the execution of the plan to be enabled. The context condition is the conjunction (denoted by "&", the logical AND) of a set of (sub-)conditions that must (all) hold to enable the plan. This sub-condition can be, at the simplest, a belief with parameters (again, ground terms or free variables), or a more complex Boolean expression over suitable variables, specified using the Python `lambda` construct to define a functor that returns a Boolean value.[3]

Finally, the *body* of a plan contains a comma-separated list of statements, enclosed in squared brackets. Each statement can be:

1. A user-defined *atomic action* (introduced above) to be executed.
2. The built-in action `set_stage(`*name*`)`, which causes the PROFETA execution environment to enter the named stage.
3. Python code (e.g., an assignment or math expression) enclosed in double quotes; note that this code can access the scope of the plan, i.e., the values taken by bound variables.
4. The assertion or retraction of a belief (+*bel* and −*bel*).
5. The request to achieve a specific goal (*goal*).
6. The request to abandon (i.e., fail) the current goal (−*goal*).

As an example, the listing in Fig. 2 shows two PROFETA plans, excerpted from the case study reported in Section 6.

Line 1 defines a *goal plan*; it means that goal `pick_new_pallet()` is to be achieved by executing the action `set_stage("area-scan")`, whereby the program enters the `"area-scan"` stage. Line 3 opens the definition of the stage named `"area-scan"`: all plans declared after this statement and before `stage("pick")` will be eligible for possible activation only after, at runtime, the program has entered stage `"area-scan"`, via a `set_stage("area-scan")` statement. Stage `"area-scan"` begins with the so-called (reactive) *stage startup plan* (line 4): indeed, entering a stage causes the automatic assertion of the `start()` belief,[4] which can be used, like here, to trigger an optional startup plan. Line 6, still within stage `"area-scan"`, has a reactive plan, whose body is triggered when a belief of the form `pallet(`t`,`u`)`, for grounded t, u, is asserted; if this happens, variables `"X"` and `"Y"` become bound respectively to t and u, and two actions are executed: stopping the robot and entering stage `"pick"`. In the latter, the startup plan (lines 9 and 10) is executed under a context condition, i.e., provided the `pallet(`t`,`u`)` belief is present in the knowledge base for some grounded t, u; in this case too, `"X"` and `"Y"` will be bound to t and u when actions following » are executed (notably, this happens for action `forward_slow("X")`).

4.3. *Structure of a* PROFETA *program*

The basic organization of a PROFETA program is illustrated in Fig. 3. The program consists of a sequence of parts. Part A lists the necessary module imports. Parts B through D contain the definition of the entities used in the program (beliefs, goal, actions and sensors). *Beliefs* and *goals* must be defined as subclasses of, respectively, **`Belief`** and **`Goal`** – these classes are provided by the framework. Likewise, actions and sensors must be defined as subclasses of **`Action`** and **`Sensor`** respectively. An action should override the `execute()` method with the code that acts on the environment. A sensor overrides the `sense()` method to specify how it senses the environment; the return value of the `sense()` method can be either `None` or a belief to be asserted in the knowledge base (which, as said earlier, could trigger the execution of a plan).

Once all entities have been specified, the PROFETA engine is instantiated and started in part E. The `PROFETA.start()` method is invoked to create all the structures required for the runtime representation of the *knowledge base* and the *plans*. The organization of these structures, from a software architecture point of view, as well as their role in the execution of a PROFETA program, will be covered in Section 5.

[3] The functor does not have parameters and can access the bound variables of the plan.

[4] Indeed, this is a special belief called *reactor*, see Section 5.

```
# ------------- PART A: IMPORTS
from profeta.main import *
from profeta.lib import *

# ------------- PART B: definition of beliefs and goals
class belief1(Belief): pass
class belief2(Belief): pass

...

class goal1(Goal): pass
class goal2(Goal): pass

# ------------- PART C: user-defined actions
class action_a(Action):
    def execute(self):
        # ... python code for action_a
...

# ------------- PART D: definition of sensors
class sensor_x(Sensor):
    def sense(self):
        # ... perform environment sensing
        return +belief1(1,2,3,4,...)
...

# ------------- PART E: instantiation of the PROFETA engine
PROFETA.start()

# ------------- PART F: plans of the PROFETA program
+belief1("X","Y") / belief2("X") >> [ action_a("X", "Y"), goal1() ]
goal1() / belief2("X") >> [ -belief2("X") ]

# ------------- PART G: adding sensors and initial beliefs
PROFETA.add_sensor(sensor_x())
PROFETA.assert_belief(belief2(200,400))
PROFETA.assert_belief(...)

# ------------- PART H: execute everything
PROFETA.run()
```

Fig. 3. Structure of a PROFETA program.

Plans are defined in section F, organized in stages when needed. As mentioned earlier, like all PROFETA code, they are also valid Python expressions. This fact, which is obvious for PROFETA entities occurring in previous program sections, also holds for plans, for PROFETA exploits *operator overloading* to redefine the following operators of its basic classes: "+", "-", "/", "&" and ">>", i.e., those used to construct plans. The execution of these overloaded operators results, at runtime, in the execution of code that builds structures which represent the plans. This enables the Python virtual machine to interpret these structures as expected according to PROFETA semantics. Further details on how they are handled by the PROFETA core are provided in the following section.

The next section, part G, declares which *sensors* will be used (these must have been previously defined in part D), and populates the knowledge base of the robot with beliefs assumed to hold. These initializations are performed by invoking PROFETA functions `add_sensor()` and `assert_belief()`, respectively. Any other system initialization—if needed—has to be performed before entering the main PROFETA loop, i.e. before invoking the `run()` function. This function never returns and continually performs sensor polling, event detection, and plan selection and execution, as shall be seen in the next section.

5. PROFETA: software architecture and implementation

5.1. Architecture of the PROFETA platform

PROFETA is structured according to the object-oriented architecture illustrated in Fig. 4.

PROFETA classes can be divided into two groups: classes above the dashed line in Fig. 4 are internal and implement the BDI reasoning engine; classes below the line, i.e. `Action`, `Sensor`, `Belief` and `Goal`, are made available to the programmer for subclassing. In the set of exported classes, additional (sub-)classes appear: `AsyncAction`, `AsyncSensor`, `SingletonBelief` and `Reactor`. These implement certain specialized aspects of their respective parent classes. Specifically, `AsyncAction` and `AsyncSensor` are used when the action or the sensing must be performed *in parallel* with respect to the execution of the plans; an `AsyncAction` is used when action termination does not affect the execution of the plan and that action requires a certain time to complete, thus its synchronous execution would introduce latencies in

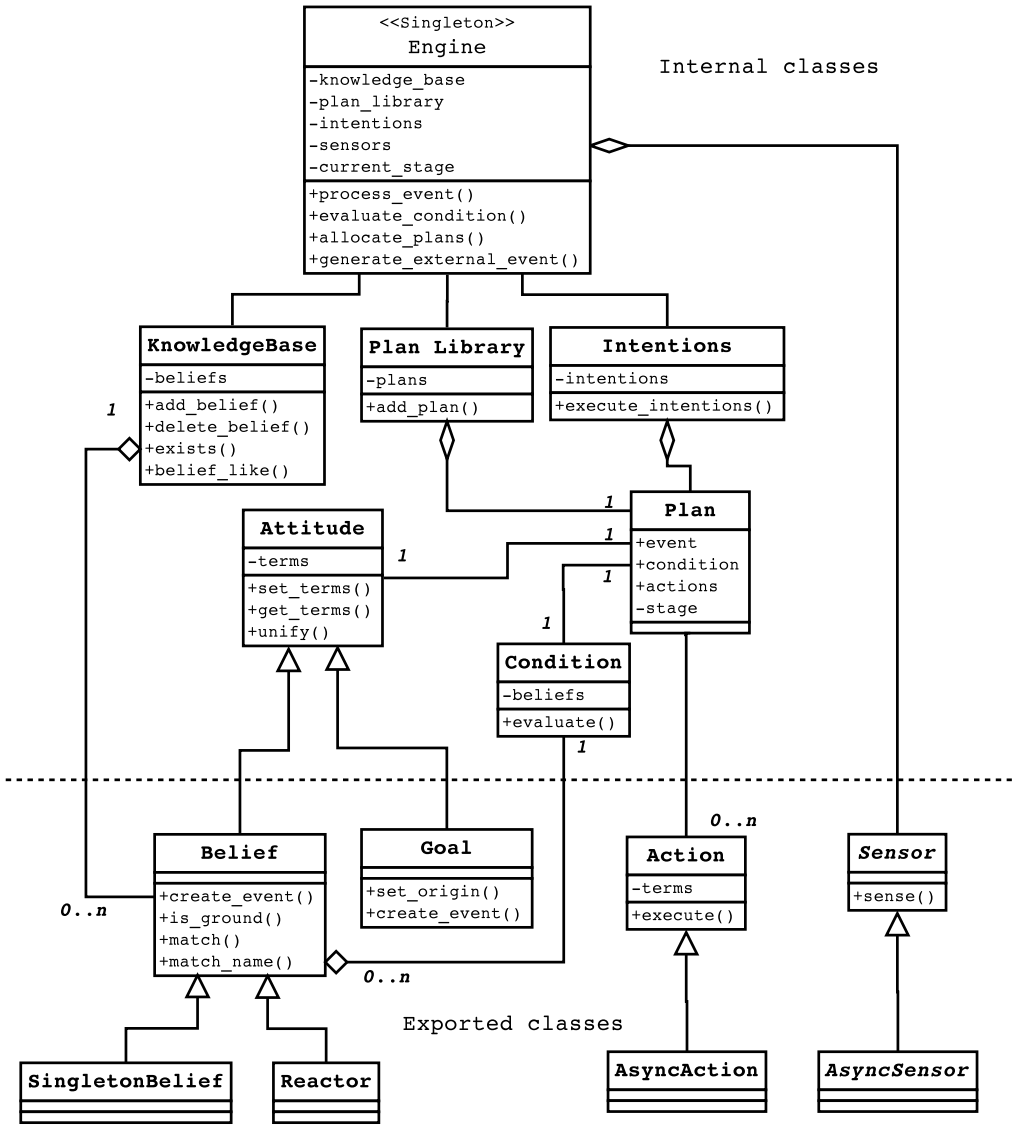

Fig. 4. PROFETA class diagram.

plans; an `AsyncSensor` is instead used when a sensor must be polled with a specific timing/dynamics. `SingletonBelief` is a belief which can exist *in a single instance* in the knowledge base; therefore, if an assert operation is performed and another belief of this type is already present in the knowledge base, that belief is replaced and not added as in normal instances of `Belief`; it is used when a state information must be handled,[5] and *update operations* on it are needed; using a `SingletonBelief` optimizes the operations since it avoids removing and reasserting the belief, as it happens in other similar platform. A `Reactor` is instead a special kind of belief which is *automatically removed* from the knowledge base when an associated plan is executed; reactors are mainly used to represent one-shot events whose effect disappears when the event itself is consumed.

`Attitude`, `Belief` and `Goal` provide methods that overload the behavior of operators "`+`", "`-`", "`/`"; `Plan` redefines operator "`>>`"; and operator "`&`" is redefined in both `Condition` and `Belief`.

These overloaded methods help to implement PROFETA's declarative semantics underlying the plan definition paradigm.

The description of a plan is encapsulated by the `Plan` class, which contains, as attributes, references representing plan components, i.e.:

- the *trigger event*, which is an `Attitude` (i.e., a `Belief` or a `Goal`),
- the *context condition*, represented by a `Condition` object, which, in turn, is a collection of `Beliefs`,

[5] A typical use of a SingletonBelief is, for example, to store the current pose of a robot.

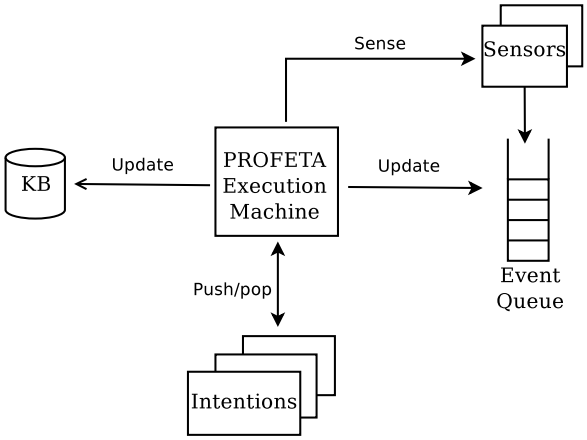

Fig. 5. PROFETA engine: PEM, sensors, event queue, knowledge base.

- a list of `Actions`,
- the associated *stage*, if any (this attribute is set to `None` if the plan is global).

Plans are collected and managed by the `PlanLibrary`, an attribute of the main PROFETA `Engine`. This is a Singleton class which includes the PROFETA Execution Machine, i.e., the main code governing the execution of a PROFETA program, as detailed in Section 5.2.

When the triggering event of a plan occurs, provided its context condition evaluates to true, that plan is selected for execution, i.e., is placed into a collection holding all plans ready to be (and not yet) executed. In BDI terminology, such a plan is an *intention* and, for this reason, the class of the collection of ready plans is called `Intentions`.[6]

Finally, the `Engine` contains an instance of class `KnowledgeBase`, which exposes methods to: *(i)* modify the set of beliefs, *(ii)* test the presence of a particular belief, *(iii)* query for a specific subset of beliefs.

5.2. *Execution of a* PROFETA *program*

As stated in Section 4.3, a PROFETA program is started by the invocation of the `PROFETA.start()` method, which triggers the instantiation (and initialization) of the PROFETA `Engine`. The main loop of the program is started through a call to the `PROFETA.run()` method, which starts the PROFETA *Execution Machine* (PEM), embedded into the `Engine` and whose functioning is illustrated in the following.

The PEM contains data structures representing the *state* of the machine itself:

- KB, the *knowledge base*, containing the asserted beliefs
- PS, the *set of plans* defined in the PROFETA program
- S, the *set of sensors* used in the program
- EQ, the *event queue*, keeps track of events that need to be processed
- STG, the *current stage*
- IS, the *intention list*, i.e., a list of plans selected for execution

These entities and their relationships are reported in Fig. 5, while the basic behavior of the PEM is exemplified by the pseudo-code of Algorithm 1 and described below.

EQ implements a queue and is manipulated by means of **enqueue/dequeue** operations. IS is a multi-purpose structure which contains the intentions and is treated as both a stack (with **push** and **pop** operations) and a queue. An intention is represented, in the PEM, as a tuple $(evt, cond, aseq)$. The first element of IS (which is the head of the queue or the top of the stack) is called *current intention* and is the one currently executed.

As specified by Algorithm 1, the execution proceeds as follows. At the start-up phase, EQ and IS are empty, and STG is set to `None`; KB could contain some initial beliefs (if they are asserted before starting the program), PS contains the program and S contains the set of sensors defined. The main execution loop of the PROFETA machine is subdivided in three phases: *sensor polling* (lines 5–10), *event processing* (lines 11–17), *intention processing* (lines 18–56).

Sensor polling is performed by scanning the S set and calling the `sense()` method of each defined sensor (line 6). The outcome of such a call can be *nothing* or the *assert/retract* of a belief or reactor. In the latter case, KB and EQ are updated

[6] As detailed later on, if multiple plans could be selected (because they are triggered by the same event and all their contexts are true), only the first plan occurring in the source code becomes an intention, i.e., enters the collection of ready plans, awaiting execution.

Algorithm 1

```
1:  EQ ← ∅
2:  IS ← ∅
3:  STG ← None
4:  while true do
5:    for all sensor ∈ S do
6:      evt ← sensor.sense()
7:      if evt ≠ nil then
8:        update(KB, EQ, evt)
9:      end if
10:   end for
11:   if EQ ≠ ∅ then
12:     evt ← dequeue(EQ)
13:     i ← find_first_intention(evt, PS, STG)
14:     if i ≠ nil then
15:       enqueue(IS, i)
16:     end if
17:   end if
18:   while IS ≠ ∅ do
19:     current ← head(IS)
20:     (evt, cond, aseq) ← current
21:     a ← remove_head(aseq)
22:     if aseq = ∅ then
23:       pop(IS)
24:     end if
25:     if a is_a action then
26:       if a is_a set_stage then
27:         STG ← a.parameter[0]
28:         enqueue(EQ, +start())
29:       else
30:         a.execute()
31:       end if
32:     else if a is_a assert then
33:       update(KB, EQ, a)
34:     else if a is_a retract then
35:       update(KB, EQ, a)
36:       update_intentions(IS, a)
37:     else if a is_a achieve_goal then
38:       i ← find_first_intention(a, PS, STG)
39:       if i ≠ nil then
40:         push(IS, i)
41:       end if
42:     else if a is_a fail_goal then
43:       pop_until(IS, a)
44:       i ← find_first_intention(a, PS, STG)
45:       if i ≠ nil then
46:         push(IS, i)
47:       end if
48:     end if
49:     if any failure during action execution then
50:       g ← pop_until_goal(IS)
51:       i ← find_first_intention(−g, PS, STG)
52:       if i ≠ nil then
53:         push(IS, i)
54:       end if
55:     end if
56:   end while
57: end while
```

accordingly (line 8). In more detail, the semantics of `update()` is as follows: if the operation is the assertion of a belief, the KB is updated and the relevant event is added to the queue EQ, unless the belief itself is a *singleton* and an instance is already present.

If a belief is instead retracted, given that it belongs to KB, the relevant event is added to EQ; however, if EQ already contains an event related to the *assertion* of that belief, the latter event is simply removed from the queue.[7]

After scanning the set of sensors, the PEM checks for the presence of an event evt in the EQ (line 11) and, if this is the case, it find the first *matching plan* (i.e., candidate *intention*). The latter, if present, is then added (enqueued) to the intention

[7] This may happen if a belief b is first asserted, and subsequently retracted before the $+b$ event is processed.

list IS (lines 13–16). Note that the concept of "first" is related to the order in which the plans are defined in the PROFETA program.

With line 18, execution of intentions begins, and goes on until structure IS becomes empty (i.e. all intentions have been executed). The first intention on the list IS (i.e. the *current intention*) is picked,[8] and the next action to execute, a, is extracted from the action list *aseq*; if *aseq* becomes empty, the intention is removed from IS. Action a is now executed (lines 25–48), in a fashion depending on its type, which may fall into one of three main categories: *atomic actions*, *event-related actions* and *goal-related actions*:

- If a is an **atomic action**, it is directly executed (lines 25–31). A special case is the built-in action `set_stage`, which changes the current stage STG according to its parameter; in addition, the built-in reactor `start()` is added to EQ; this event can be used to automatically trigger a start-up plan (if any) for the entered stage.
- The second category (*event-related actions*) is relevant to belief assert and retract operations, which basically cause the addition, to EV, of the relevant $+b$ or $-b$ event:
  - If a is a **belief assert**, the KB is updated accordingly and the relevant event is added to the event queue (lines 32–33).
  - A similar operation is performed when a refers to **belief removal**, of the form $-b$; in addition, IS is searched for the presence of an intention related to event $+b$: if present, this intention will be removed from IS, for the "fact" which caused it to be scheduled for execution no longer holds (lines 34–36).
- The third category (*goal-related actions*) includes actions pertaining to *goal achievement* and *goal abandonment* (or goal failure); executing them essentially amounts to *push* or *pop* intentions as appropriate:
  - Goal achievement, is treated by searching for a plan matching the requested achievement operation (line 38); if such a plan is found, it is transformed into an intention which gets *pushed* onto the intention list (39–41); the push operation is required since a goal achievement is—more or less—akin to a procedure call of an imperative language;
  - A goal failure request—goal abandonment—is executed by canceling any pending execution of the same goal to be abandoned (lines 42–48), i.e., by popping, from IS, all intentions until the request to achieve the goal to be abandoned is found ($pop\_until(IS, a)$). Subsequently, the engine searches for a plan that the current, goal-failure, event a could trigger; if present, it is made an intention and pushed onto IS (lines 44–46).
- When the execution of the current intention fails for any reason, i.e., an unhandled run-time error is encountered,[9] a *goal failure* event must be generated; for this purpose, intentions are popped from IS until the first goal achievement request, say g, is found (lines 49–56); subsequently, event $-g$, amounting to failure/abandonment of g is generated, a relevant plan is searched for and, if present, made into an intention and pushed onto IS.

6. Case study

PROFETA—whose current version is available on GitHub[10]—has been used by the authors of this paper to write the software for several of the robots built in their laboratory. In particular, PROFETA has been used to program robots participating to the Eurobot competition.[11] The aim of this competition is to build a mobile robot capable of collecting, manipulating and sorting a range of different objects, usually distributed across a playing area. The challenge of this competition is manifold: robots must operate in an autonomous manner, i.e. without any kind of external aid; two or more opponent robots might be operating in the same area, thus requiring mechanisms for localization and collision avoidance; opponents can move objects from their original locations on the playing area, thus robots must be able to cope with fast-changing conditions. These aspects make the Eurobot competition an interesting test-bed for the proposed framework: despite the complexity of the game, PROFETA enabled us to write game strategies in a simple, declarative manner; at the same time, the reasoning engine offered by PROFETA allows robots to dynamically adapt their behavior during the game, in response to perceived events [65,66].

While our previous work demonstrated the use of PROFETA in the context of a robotic competition, here we provide a case study involving an industrial scenario—the objective is to illustrate how PROFETA can be used in a more realistic application. To this aim, we will consider a sample application involving a forklift robot moving boxes in a warehouse.

Fig. 6 depicts the application scenario: the task of the robot (illustrated in Fig. 7) is to move pallets (one at a time) from their initial locations (*c1*, *c2*, ... *c8*) to one of the depots (*dep1*, *dep2*). Each initial location may or may not contain a pallet, therefore the robot has to scan the area through presence sensors before picking a pallet from it. Each pallet has type that the robot can detect (using, e.g., bar-codes, QR-codes or RFID tags). Selection of the final depot area for a pallet is based on its type; moreover, we further assume that the association between pallet type and the depot is dynamic and can be established at run-time through suitable beliefs like `bring_to(PalletType, Depot)`. Once all pallets have been moved,[12] the robot finally goes back to its parking area, which is represented by point *park* in Fig. 6.

[8] It is only read and *not extracted*.

[9] This correspond to the occurrence of an uncaught exception from the point of view of the Python language.

[10] https://github.com/corradosantoro/profeta/.

[11] http://unict-team.dmi.unict.it, http://www.eurobot.org.

[12] We suppose that each depot is able to host more than a pallet.

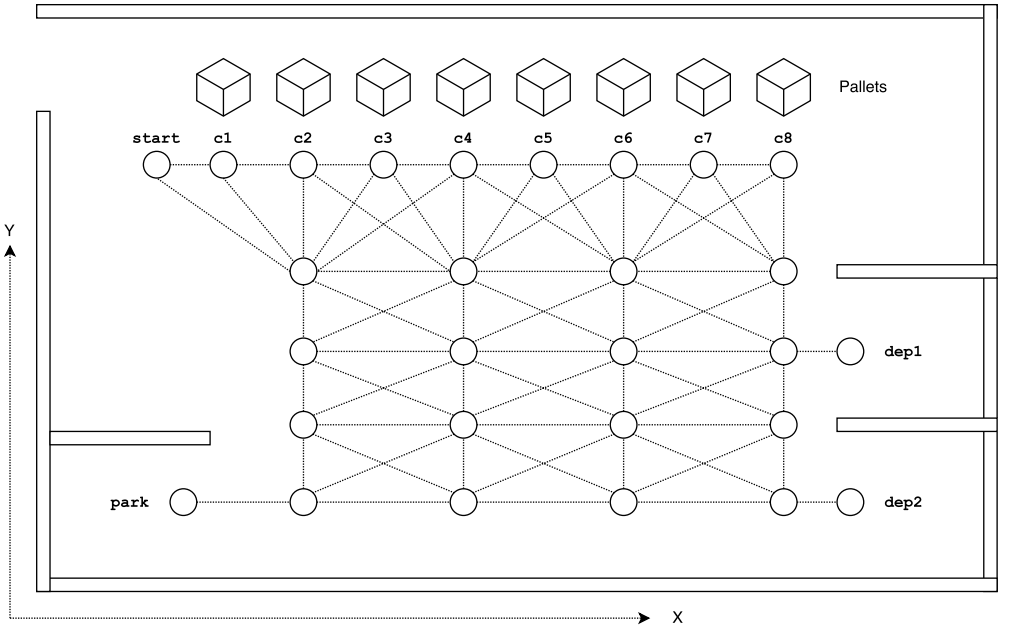

Fig. 6. Forklift Case study. A forklift robot moves pallets from one location to another in a warehouse, according to a specific plan. The robot moves to the pallet area, from which it picks a pallet at time and move it to one of the three depots, on the basis of the content.

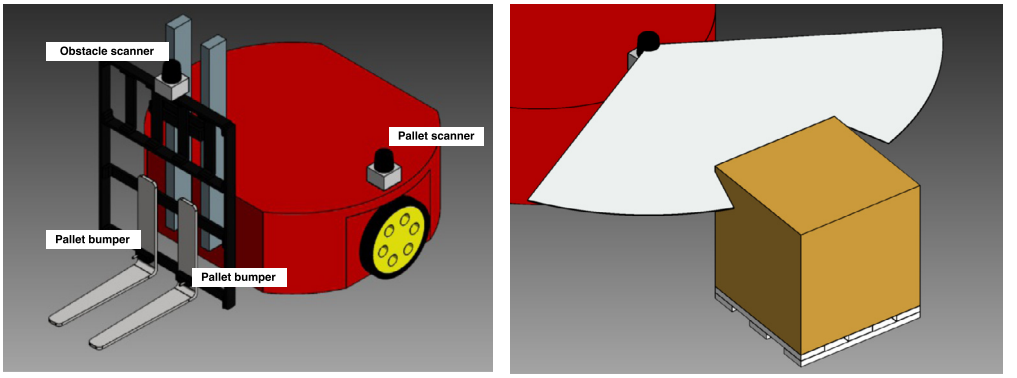

Fig. 7. A 3D model of the forklift robot considered in the case-study (left-side) and a detail of the beam scanner sensor (right-side).

In our scenario, the robot shares the environment with other humans and machines, so it must implement a collision avoidance strategy. The possible paths that can be taken by the robot are constrained to belong to the graph depicted in Fig. 6 using dashed lines for the edges. Dijkstra's shortest-path algorithm is used to determine the road to follow from a starting point (e.g. *c3*) to a destination (e.g. *dep2*); should an obstacle be detected during the cruise, the same algorithm is used to determine an alternative path. Beside graph-based navigation, we also postulate that other primitives are available to make the robot perform make simple movements, like rotating a certain amount of degrees, going forward a certain distance, reaching a certain point through a rotate-and-go-straight motion, etc. Given these navigation approaches, we suppose that a localization system is present in the environment, so that the robot is able to detect its own location, in a XY coordinate space, and use it to navigate.

The behavior of the robot can be implemented using the strategy outlined in Table 1.

In step 1, the robot reaches point *start*, then scans each station from $c1$ to $c8$ (step 2) until a pallet is found; during this step, a sensor (i.e. a LIDAR), placed on the left side of the forklift, is used to detect the pallet (see the 3D sketch in the right side of Fig. 7). Once a pallet is found, it is first picked (step 3), and then its type is detected (step 4) to determine where it should be moved; the move operation is performed in step 5; at its end, after releasing the pallet, the program goes back to step 1. If no pallet is found during step 2, the robot goes back to the parking location (step 6).

From a PROFETA point of view, this behavior can be implemented using four stages, namely *area-scan*, *pallet-pick*, *depot* and *parking*. The association between stages and the step/tasks described is reported in Table 1.

Table 1
Description of the forklift behavior.

Step	Description	Tasks	Stage
1	Approaching the pallets area	Move towards point *start*	`area-scan`
2	Scanning the pallets area	Move towards point *c*8 with pallet scanner activated. If a pallet is found go to step 3, otherwise go to step 6.	
3	Picking the pallet	Drive towards the pallet and actuate the forks	`pick`
4	Identifying the pallet	Activate the RFID/QRCode reader	
5	Carrying the pallet to depot	Move towards a point *depX* and then release the pallet	`to-depot`
6	Going to parking	Move towards point *park*	`to-parking`

Table 2
Beliefs used in the case-study.

Type	Name	Meaning	Asserted by
Singleton Belief	moving_to(T)	Forklift is moving to a target T	*Plan*
	pose(X, Y, Theta)	The current pose of the robot	PoseSensor
	pallet(X, Y)	The relative position of a pallet	PalletScanner
	pallet_type(P)	The type of the pallet identified by forklift sensors	PalletIdentifier
Reactor	path_completed()	The forklift has reached the destination point of a path	PathSensor
	obstacle()	Forklift has encountered an obstacle	ObstacleScanner
	bump()	One of the fork bumpers is active	ForkBumpers
	lift(L)	The current height of the lift	LiftSensor

The robot is equipped with three main sensors (see Fig. 7): *(i)* a LIDAR to detect pallets, *(ii)* another LIDAR, placed in front, to check for heads-on obstacles, and *(iii)* two bumpers placed in the forks to understand when the pallet is ready to be lifted. Table 2 shows how we map onto beliefs data from these physical sensors, as well as other state and environment information managed by plans (cf. `moving_to()`).

Physical sensors are accessed by means of the following `Sensor` classes: `PathSensor`, `PoseSensor`, `ObstacleScanner`, `PalletScanner`, `PalletIdentifier`, `ForkBumpers` and `LiftSensor`. The first two sensors have the task of interacting with the underlying motion system: `PathSensor` asserts the reactor `path_completed()` when the robot has reached an established target position, while `PoseSensor` asserts the singleton belief `pose(X,Y,Theta)` that represents the current pose of the robot.[13] `ObstacleScanner` polls the front LIDAR and, whenever detects the presence of an obstacle, asserts reactor `obstacle()`. Pallet scanning and localization is performed by means of the side LIDAR, which is interfaced through `PalletScanner`; this PROFETA sensor does not only read LIDAR scans, but also performs some processing of the data, aiming at recognizing a shape with its beam, as illustrated in the right side of Fig. 7; on this basis, it computes the position $\{X_p, Y_p\}$ of the pallet with respect to the robot and makes it known for PROFETA code by asserting belief `pallet(X,Y)`.

Pallet type is instead read by `PalletIdentifier` which, once activated by a RFID or QR-code reader, asserts belief `pallet_type(P)` accordingly. Sensor `ForkBumpers` has the task of polling the bumpers present in the forks and, when one of it is activated, assert the reactor `bump()`. Finally, sensor `LiftSensor` polls the lift motion system and asserts the belief `lift(L)` where `L` represents the current height of the lift from the ground (in cm); this belief is treated as a reactor since it is used to trigger a plan which stops the lift as soon as it reaches a certain position[14] (see, e.g., line 30 in Program 1).

Actions used in this case-study are summarized in Table 3; they are classified into three main categories.

The *driving* (or *motion*) *actions* control the robot movements; they only initiate motion, i.e. *trigger* the motion, but do not wait for the robot to reach the target point—this aspect is handled by `PathSensor`.[15] The first two actions, `dijkstra_move_to(T)` and `dijkstra_move_to_excluding(T,X,Y)` are used to move the robot towards a node `T` of the graph by planning the shortest path using the graph itself; the latter action takes, as additional parameter, a point (X, Y) that must be *excluded* in the planning: as it will be shown in the following, this is required when an obstacle is met and thus the relevant point (occupied by the obstacle) must be excluded in the next planned path. The `move_to(T)` is instead used to move the robot towards a node `T` of the graph but a straight path. The other driving actions—`rotate_to(A)`, `forward_slow(D)` and `stop_robot()`—handle simple movements of the robot in the XY plane and of course do not use the graph. The second category of actions are those related to *sensor handling*. Note that some sensor classes should perform data polling only when needed (rather than in any case); for this reason, we introduce, for such sensors, actions to turn them on (or off) as required by plans. The third category of actions *control the lift* and are used to start lift movement

[13] The pose of a robot in a 2D environment is represented by the triple $\{X, Y, \theta\}$, where X and Y are the coordinates of a robot's reference point and θ is the heading of the robot.

[14] We are assuming the hardware driving the lift **does not** support position control.

[15] This aspect is elaborated in more details later on, when the code of the case-study is analyzed.

Table 3
Actions used in the case-study.

Type	Name	Meaning and parameters
Motion Actions	dijkstra_move_to(T)	Triggers the movement of the forklift towards target T using the graph and Dijkstra's algorithm
	dijkstra_move_to_excluding(T,X,Y)	Triggers the movement of the forklift towards target T using the graph and Dijkstra's algorithm, but excluding the graph point nearest to (X, Y)
	move_to(T)	Triggers the movement of the forklift towards target T **without** using the graph (straight path)
	rotate_to(A)	Triggers a rotational movement in order to head to absolute orientation A (in degrees)
	forward_slow(D)	Triggers a forward slow motion of a distance D (in meters, negative values allowed)
	stop_robot()	Halts the forklift
Sensor Handling Actions	activate_scanner()	Turns on the sensor `PalletScanner`
	stop_scanner()	Turns off the `PalletScanner`
	identify_pallet_type()	Identifies the type of the goods in the pallet
	activate_bumpers()	Turns on the sensor `ForkBumpers`
Lift Actions	lift_up()	Turns on the lift to go up
	lift_down()	Turns on the lift to go down
	lift_stop()	Stops lift motion
Other Actions	wait_seconds(S)	Waits for S seconds

(up or down), and to stop it. Among *other actions* (fourth category), we find only `wait_seconds(S)`, which halts the program for `S` seconds.

All driving actions are *asynchronous*, i.e. they instruct the underlying layers and/or the hardware to perform that action but **do not wait for completion**, which, in this case, would entail waiting for target achievement. Indeed, that such a driving action fails of succeeds is supposed to be captured by a proper Sensor (by polling the underlying layer), and then made known to the PROFETA code by means of a suitable belief.[16] On this basis, a task like "moving to a target position *T*" cannot be handled in a single plan but is spread over several ones: first the moving action is triggered; later, when `path_completed()` is asserted, the target is considered to be reached and the next task of the behavior can be executed; however, if, during motion, an obstacle is detected (cf. belief `obstacle()`), an avoidance policy is first deployed, and the motion action is then re-triggered.

In the forklift application (as in many other robotic applications) there are several motion tasks like the one illustrated above, so, in order to avoid code duplication, a certain form of code encapsulation and reuse should be employed. This is readily achieved in PROFETA for, thanks to some characteristics of the Python language, plans can be parameterized in all of their parts.

As an example of this, consider parameterized function `drive_and_avoid()` in our case study (cf. Program 1, lines 5–14). Its parameters are: the PROFETA action `move_action` which triggers motion, the `target` of the motion, and the list of PROFETA actions, `next_actions`, to be performed once the target is reached. At lines 22 and 23 `drive_and_avoid()` is "instantiated" twice, with different "actual" parameters, `dijkstra_move_to` vs. `move_to`, to obtain two separate goals handling path selection in different ways. Also noteworthy how, by inspecting its `move_action` parameter, `drive_and_avoid()` manages to differentiate the obstacle avoidance policy: for Dijkstra's algorithm motion, it tries to find a different path (lines 9–11) by ignoring the point of the graph nearest to the obstacle itself, whereas for straight path motion, it waits for a certain amount of time before retrying (lines 13–14).

Highlighting the power of the PROFETA approach, the forklift is implemented as a convenient mix of the imperative and declarative programming paradigms: while the main behavior is programmed in terms of PROFETA plans, the path planning algorithm is coded imperatively. Listings 1 and 2 document these two styles, respectively. In them, some uninteresting details have been omitted (about entities/classes declaration or library imports) to focus instead on the most significant bits.

The main goal of the program is `pick_new_pallet()` (Listing 1, line 18) which is initially triggered at program startup (line 54); its sole action is to enter stage `"area-scan"` which, in turn, triggers the stage startup plan in line 21. The task required here is to let the robot reach point *start* (see Fig. 6, page 47); this is performed by the plans defined by function `drive_and_avoid`, triggered by goal `drive_to(target)`, actually `drive_to("start")` in this instance. Subsequently, again exploiting function `drive_and_avoid`, the robot moves to point *c8* through a straight path, with the pallet sensor active (lines 23–24); if the sensor recognizes a pallet, the path is stopped (line 24) and the program enters stage `"pick"`; on the other hand, if scanning is completed without detecting any pallet, target *c8* is reached (line 22), thus the robot, which has no more pallets to pick, should go to parking by entering stage `"to-parking"`.

[16] As it will be discussed in Section 7, assessing the pros and cons of asynchronous programming, this arrangement is required to avoid blocking/synchronous software actions, which, in a robotic environment, could be harmful.

Program 1 A declarative PROFETA program to drive a forklift robot.

```
1  # ... profeta imports
2  # ... beliefs, actions, goals and sensor definition
3  from dijkstra import *
4
5  def drive_and_avoid(move_command, target, next_actions):
6      drive_to(target) >> [ move_command(target), +moving_to(target) ]
7      +path_completed() / moving_to(target) >> next_actions
8      if move_command == dijkstra_move_to:
9          +obstacle() / (moving_to(target) & pose("X", "Y","_") ) >> \
10               [ stop_robot(),
11                 dijkstra_move_to_excluding(target, "X", "Y") ]
12     if move_command == move_to:
13         +obstacle() / moving_to(target)  >> [ stop_robot(), wait_seconds(30),
14                                               drive_to(target) ]
15
16 PROFETA.start()
17
18 pick_new_pallet() >> [ set_stage("area-scan") ]
19
20 stage("area-scan")
21 +start() >> [ drive_to("start") ]
22 drive_and_avoid(dijkstra_move_to, "start", [activate_scanner(),drive_to("c8")] )
23 drive_and_avoid(move_to, "c8", [stop_scanner(),set_stage("to-parking")] )
24 +pallet("X","Y") >> [ stop_robot(), set_stage("pick") ]
25
26 stage("pick")
27 +start() / pallet("X", "Y") >> [ rotate_to(90), forward_slow("X"),
28                                  activate_bumpers() ]
29 +bump() >> [ stop_robot(), lift_up() ]
30 +lift("P") / (lambda : P >=50) >> [ lift_stop(), identify_pallet_type(),
31                                     set_stage("to-depot") ]
32 +path_completed() >> [ alarm() ]
33
34 stage("to-depot")
35 +start() / (pallet_type("T") & bring_to("T", "D"))  >> [ drive_to("D") ]
36 +start() >> [ alarm() ]
37 drive_and_avoid(dijkstra_move_to, "X", [ lift_down() ])
38 +lift("P") / (lambda : P <= 5) >> [ lift_stop(), forward_slow(-2),
39                                    set_stage("area-scan") ]
40
41 stage("to-parking")
42 +start() >> [ drive_to("p") ]
43 drive_and_avoid(dijkstra_move_to, "p", [ ])
44
45 # ... sensor add
46
47 # ... initial beliefs
48
49 PROFETA.add_belief(bring_to("pallet-type-a","dep1"))
50 PROFETA.add_belief(bring_to("pallet-type-b","dep2"))
51 PROFETA.add_belief(bring_to("pallet-type-c","dep2"))
52 PROFETA.add_belief(bring_to("pallet-type-d","dep1"))
53 # ....
54 PROFETA.achieve(pick_new_pallet())
55 PROFETA.run()
```

In stage `"pick"`, the plans have the aim of making the forklift pick the pallet. In a real application, this task may be quite complex since the robot should adequately move the forks in order to perform a correct alignment with the pallet holes to ensure the right picking. Here, we consider a simplified environment and assume that the `PalletScanner` detects the pallet when the robot is properly aligned; thus, actions to be taken are: *(i)* rotate to 90 degrees (absolute heading), *(ii)* go forward at a low speed, and *(iii)* activate the fork bumpers (lines 27–28); if bumpers are hit *before* motion completion, this means the pallet has been correctly forked and needs to be lifted up (line 29). On the other hand, if the motion path is completed without bumping the pallet, something wrong happened, and the robot should stop itself and raise an alarm (line 32).

When bumpers are hit, the forks are lifted up (line 29), until the position of the lift has reached a threshold set to 50 cm (lines 30–31); if this happens, the lift is stopped, the type of the pallet is identified and the program enters stage `"to-depot"`.

In this stage, the first plan (line 35) determines first which depot the lifted pallet should go to, by means of the applicable `bring_to(`T`,`D`)` belief, specifying that depot D is where pallets of type T belong. The robot is then driven towards the target depot (line 37), and the lift is lowered to the height of 5 cm, in order to release the pallet (lines 38–39); as a

Program 2 The imperative path planner for the forklift robot.

```
from profeta.action import *

# ... other imports

class Djikstra:
    def path_to(self, point, excluded_point = None):
        # .... Djikstra's algorithm
        return path # a list of pairs (X,Y)


dijkstra = Djikstra()

class dijkstra_move_to(Action):       # a subclass of Action
    def execute(self):
        for point in dijkstra.path_to(self[0]):
            move_robot_to(point[0], point[1])

class dijkstra_move_to_excluding(Action):   # a subclass of Action
    def execute(self):
        for point in dijkstra.path_to(self[0], (self[1], self[2])):
            move_robot_to(point[0], point[1])
```

consequence, the robot goes backward two meters (the adequate distance to fork-off the pallet) and restarts its operation by entering stage `"area-scan"` again.

To complete the description of the case-study, we illustrate the chief aspects of the imperative part of the case study code. Listing 2 shows class `Dijkstra`, which implements the path planner, and the classes implementing actions `dijkstra_move_to()` and `dijkstra_move_to_excluding()` (it will be recalled that actions should be subclasses of `Action` and override its `execute()` method).

For `dijkstra_move_to()`, `execute()` invokes the planner, which returns a list of points that should be traversed by invoking the (low-level) motion subsystem's `move_robot_to` function.[17]

For action `dijkstra_move_to_excluding(t,x,y)`, `execute()` instructs the planner to find a path by ignoring the point of the graph with is the nearest to (x, y): this is required to bypass an obstacle by ensuring that point will not be selected in the path.

7. Discussion

After having presented PROFETA, in this section we review its characteristics and provide a comparison with other BDI platforms.

PROFETA is inspired by AgentSpeak(L), like the other known implementation Jason [49] and eJason [69], but while these frameworks support the "pure" AgentSpeak(L) syntax and semantics (with no modifications), PROFETA introduces some differences. The syntax has been substantially modified, mainly in order to be adapted to the use of Python expressions and operators; this is a key aspect, in that the main design principle of PROFETA is to let a programmer use, within the same software development environment, both the *imperative* and *logic/declarative* programming paradigm. While this concept is also present in other BDI platforms [70,49,52], the difference lies in the way in which such a mixing of paradigms is achieved. In this sense, there are two different approaches commonly used: A first approach is based on writing the declarative (intelligent) part using a different programming language than the underlying platform, so that it will have to be interpreted by the platform at run-time; this is the case, e.g., of Jason (and eJason) and Jadex [52]. While such an approach keeps a strong separation between the imperative and declarative parts, a feature that is desirable when the implementation of the two parts is assigned to two different teams of developers, it undoubtedly poses performance problems due to the interpreted execution of the declarative part; moreover, as the imperative part often closely interacts with the declarative one, the provided API between the two domains must be designed in such a way as to avoid additional parsing, which can be highly time-consuming, or other kind of inefficiencies.[18] The second approach is based on extending the language used to implement the platform by adding new keywords and constructs, as in the case of JACK [22] which, by means of this method, provides suitable ways to represent, in Java, the entities of the BDI model. Then, an ad-hoc pre-processor is used to translate such constructs into language lines of code (Java, in the specific case) in order to allow direct compilation and execution. This method allows an easy "mingling" of BDI construct into imperative code and does not suffer of the performance problems that interpreted approaches have, but has the drawback of requiring the ad-hoc pre-processor.

[17] It should be recalled we are assuming the motion system to handle commands in an asynchronous way.

[18] An example of such a lack of efficiency is the Jason API: here, if a belief deriving from e.g. reading a sensor must be added in the belief base, the piece of Java that performs sensor reading must prepare the relevant belief in a string that is then parsed by the Jason interpreter; this parsing is required each time the sensor is read.

On the other hand, the approach implemented by PROFETA has the advantages of the second method but without requiring the pre-processor. The support for declarative constructs is achieved by simply changing the meaning of the constructs already present in the language, in particular operator overloading, which is supported by Python. We believe that this technique enables superior run-time performance with respect to other BDI frameworks that implement the first method: since PROFETA plans are Python expression, their evaluation, from the Python point of view, causes the constructions of suitable runtime structures that allow a fast execution (from the PROFETA point of view) of the program. Investigating such performance aspects is beyond of the scope of this paper and will be the subject of future work.

PROFETA is thus a live demonstration of how the object-oriented technology can be successfully used to perform a seamless "mingling" of a logic/declarative model into a traditionally imperative programming language thus providing a multi-paradigm environment, an experience that indeed can be easily ported to other languages, rather than Python, given that they support user-defined operator overloading (for instance, C++ or Scala), which is not supported by Java [71]. At this regards, we aim to implement PROFETA in C++ in a future work.

Another important difference of PROFETA with respect to AgentSpeak(L) is the presence of several kind of beliefs, i.e. the "simple" *Beliefs*, the *SingletonBeliefs* and the *Reactors*. They have been introduced for two main reasons: better adherence to the environment modeled, and performance. As explained earlier, a *SingletonBelief* is used when we have to represent a knowledge that, according to the system to be implemented, exists in a single instance, so that any assert operation results in the modification of the already existing belief. In Jason, the corresponding operation would be "$-+bel$", meaning "remove the first instance of *bel* and the (re-)add *bel*"; however, while the resulting effect is the same, the adoption of SingletonBelief has some advantages. The first is a software engineering aspect: having a specific entity modeled in the framework affords the availability of a reference construct in the design of the system; also, in maintaining the code, the fact that a belief is single instance can be immediately understood from its declaration, while, in Jason, the same information must be derived by carefully inspecting the program code and, in particular, by looking for "$-+bel$" constructs. Moreover, the presence of an ad-hoc construct avoids programming error which, instead, are possible in Jason when a programmer "forgets" to use the "$-+bel$" operation (and uses instead the simple $+bel$). The second aspect is related to performance: the "$-+bel$" construct first removes that belief and then adds it once again (even if with different parameters), thus causing two manipulations of the knowledge base; indeed, in this case, a simple *update* operation would suffice and this is what happens in PROFETA when a $+SingletonBelief$ action is executed.

In PROFETA, we introduced Reactors to represent and handle "one shot events" that do not provide any knowledge. In PROFETA, Reactors are intended to be mainly generated by Sensors. They are not present in Jason/AgentSpeak(L), but the same semantics can be easily implemented by immediately removing the belief (using $-bel$) in the plan that handles its assertion; however, also in this case, there is a drawback in terms of performances because the knowledge base needs to be updated twice, while the use of a special belief avoids this side effect.

An additional feature provided by PROFETA is the abstraction of *stages*. As it has been reported in Section 4, stages represent specific phases of the behavior and are intended to provide a form of encapsulation of plans thus facilitating code organization and maintenance; in this sense, they are quite similarly to JACK capabilities. The use of such an abstraction, as the case-study shows, makes behavior implementation easier, in that it allows a programmer to clearly identify the plans that "make sense" in that phase. Indeed, the same objective can also be attained, without the stage abstraction, by using e.g. a (singleton) belief that keeps track of the current stage: as an example, the listing in Program 3 shows the plans of the case-study belonging to stage `"pick"`, rewriting them without the stage abstraction and using the belief `in_stage()` to track the current phase. But, while the resulting execution will be the same, there are (at least) three important aspects that must be considered; first, the concept of "encapsulation" has completely disappeared in Program 3, and the notion that those plans belong to the same behavior phase must be derived by looking at all of the plans; the second aspect concerns the readability and maintenance degree of the code which, in Program 3, is surely reduced; the third aspect relates to performance, for encapsulation in stages "helps" the reasoning engine to efficiently select plans by restricting *a priori* the set of eligible ones.

Program 3 Stages implementation without the "stage" abstraction.

```
1 +start() / (in_stage("pick") & pallet("X", "Y")) >> [rotate_to(90),
2                                                      forward_slow("X"),
3                                                      activate_bumpers() ]
4 +bump() / in_stage("pick") >> [ stop_robot(), lift_up() ]
5 +lift("P") / (in_stage("pick") & (lambda : P >=50)) >> [ lift_stop(),
6                                                          identify_pallet_type(),
7                                                          +in_stage("to-depot"),
8                                                          +start() ]
9 +path_completed() / in_stage("pick") >> [ alarm() ]
```

A feature that is present in Jason, but not provided in PROFETA, is the possibility of running *concurrent/parallel plans*. This can be achieved in Jason by running more than one goal at system start-up, or using a special operator that starts concurrent goals. This is a very interesting feature, which is definitely useful in the context of software agents. However, in PROFETA, which is specifically designed for robotics, the absence of concurrent plans is a choice by design. Indeed, in

robotics, concurrency is often referred to the possibility of running, in parallel, two or more independent *physical tasks*: as an example, a robot could execute, in parallel, the actions of *(i)* driving over a certain path and *(ii)* manipulating (by means of an arm) a certain object that it has previously picked. At first sight, such actions could require pieces of program code running in parallel; yet, trying to think up the possible plans, it is soon realized that things are not so straightforward.

As it has been stated in Section 6, the basic programming model of a robotic application is mainly *asynchronous*, meaning that the high-level program has the responsibility of "triggering the actions", which, in turn, are managed and completed by the lower layers (and often in specific, maybe proprietary, daughterboard drivers). Moreover, the program has a need to receive a feedback, from suitable sensors, which notifies the completion of the action, or its failure (so that it can react accordingly with suitable counter-measures); such a feedback is also received with a latency which is dependent on the physical environment and is order of magnitudes higher than CPU execution times.

As a solution, rather than using explicit concurrency, we can code in the fashion illustrated in Program 4. I.e., we can simply start our actions (line 1), which (being asynchronous) will be practically executed in parallel; then, we can gather the relevant feedbacks with other plans (line 4 and 5), by using the data provided by sensors. It will be understood, that this asynchronous programming style does not require the explicit presence of concurrent code: indeed, in the various robots developed in our laboratory, we never met the need for explicit concurrency in PROFETA code, which is why it has not been implemented as of the current version of the tool.

Program 4 Asynchronous actions and parallelism.

```
1 ... >> [ drive_robot_to("X", "Y"), move_arm_to("X1","Y1","Z1") ]
2        # These two actions trigger two movements that are executed,
3        # in parallel, by the underlying levels or daughter boards
4 +path_completed()       >> [ ... ] # do other things after path
5 +arm_position_reached() >> [ ... ] # do other things after arm movement
```

8. Conclusions

In this paper we described PROFETA, a framework to program the behavior of autonomous robots in a declarative way. PROFETA is based on the Belief–Desire–Intention (BDI) programming paradigm, which is mostly used in the field of autonomous agents. Our work presents some similarities with AgentSpeak(L), by which it is inspired, but it has been entirely written in Python, which affords some distinctive advantages. The proposed approach combines the power of the object-oriented paradigm, which is useful in programming robotic devices, and the declarative paradigm, which is very powerful in defining the robot's autonomous behavior.

The main contribution of our approach consists in the integration, within a single framework, of the BDI model without having to deal with the shortcomings of traditional layered architectures, where the upper layer would be implemented as a *logic system*, and the underlying layers are based on algorithms generally implemented using an *imperative language*. This juxtaposition of different programming approaches forces developers to deal with two different execution environments, and generally makes the overall software system harder to debug and maintain.

On the other hand, we have demonstrated, through a non-trivial case-study, how, within the PROFETA framework, robotic system design and development can be supported by a unique execution environment, i.e. the Python virtual machine. With PROFETA, Python object oriented/imperative constructs are available for the imperative-based components, while the robot's "intelligent behavior" can be implemented, in Python too, by means of the suitable additional constructs introduced. Dealing with a single language/environment has obvious advantages in terms of reduced codebase complexity, and improved maintainability and reliability of the overall software system, above all, when a Python IDE with debugging capabilities is used to develop the system.[19]

Unlike other frameworks, PROFETA does not provide a GUI to support designers and programmers, which could be a requirement for application scenarios where rapid prototyping is paramount. However, we are in the process of adapting to PROFETA visual design tools available for the Distilled StateCharts (DSC) formalism, which models agent behavior with Statecharts-like state machines. We have already investigated the relationship between DSC and the BDI-inspired PROFETA in [9,72].

As stated in Section 3, in the recent years, we have developed an approach named GOLEM [60], which is an abstract framework for autonomous robot programming. GOLEM represents a different approach, on which robot behaviors are designed to mimicking human behavior by organizing activities into goals and sub-goals, linked to one another through specific relationships. Although the design principles of GOLEM are very different from PROFETA, we aim to integrate—in a future work—the two approaches.

[19] like e.g. Wingware (http://wingware.com/).

Acknowledgements

This work is partially supported by projects PRISMA PON04a2 A/F, CLARA and MEDNETNA funded by the Italian Ministry of University.

References

[1] R. Siegwart, I. Nourbakhsh, Introduction to Autonomous Mobile Robots, MIT Press, 2004.
[2] R.C. Arkin, Behaviour-Based Robotics, MIT Press, 1998.
[3] R.R. Murphy, Introduction to AI Robotics, MIT Press, 2001.
[4] C. Santoro, An Erlang framework for autonomous mobile robots, in: ERLANG '07: Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, ACM Press, 2007.
[5] R.A. Brooks, A robust layered control system for a mobile robot, IEEE J. Robot. Autom. 2 (1) (1986) 14–23.
[6] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (1987) 231–274.
[7] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, Jade: a software framework for developing multi-agent applications. Lessons learned, Inf. Softw. Technol. 50 (1) (2008) 10–21.
[8] G. Fortino, F. Rango, W. Russo, Engineering multi-agent systems through statecharts-based jade agents and tools, in: Transactions on Computational Collective Intelligence VII, Springer, 2012, pp. 61–81.
[9] G. Fortino, W. Russo, C. Santoro, Translating statecharts-based into BDI agents: the dsc/profeta case, in: Multiagent System Technologies, Springer, 2013, pp. 264–277.
[10] J.M. Bradshaw (Ed.), Software Agents, AAAI Press/The MIT Press, 1997.
[11] G. Weiss (Ed.), Multiagent Systems, The MIT Press, 1999.
[12] M. Wooldridge, P. Ciancarini, Agent-Oriented Software Engineering: The State of the Art, Lecture Notes in Computer Science, 2001, pp. 1–28.
[13] N. Jennings, On agent-based software engineering, Artif. Intell. 117 (2) (2000) 277–296.
[14] A. Rao, M. Georgeff, BDI agents: from theory to practice, in: Proceedings of the First International Conference on Multi-Agent Systems, ICMAS-95, San Francisco, CA, 1995, pp. 312–319.
[15] M.E. Bratman, Intentions, Plans and Practical Reason, Harvard University Press, 1987.
[16] B.W. Kernighan, D.M. Ritchie, P. Ejeklint, The C Programming Language, vol. 2, Prentice-Hall, Englewood Cliffs, 1988.
[17] R. Hyde, The Art of Assembly Language, No Starch Press, 2010.
[18] F. Ingrand, M. Georgeff, A. Rao, An architecture for real-time reasoning and system control, IEEE Expert 7 (6) (1992) 34–44.
[19] A. Rao, AgentSpeak (L): BDI Agents Speak Out in a Logical Computable Language, Lecture Notes in Computer Science, vol. 1038, 1996, pp. 42–55.
[20] A. Pokahr, L. Braubach, W. Lamersdorf, Jadex: A BDI Reasoning Engine, Multiagent Systems Artificial Societies and Simulated Organizations, vol. 15, 2005, p. 149.
[21] Jason Home Page, http://www.jason.sourceforge.net/, 2004.
[22] N. Howden, R. Rönnquist, A. Hodgson, A. Lucas, Jack intelligent agents-summary of an agent infrastructure, in: 5th International Conference on Autonomous Agents, 2001.
[23] A. Di Stefano, C. Santoro, eXAT: an experimental tool for programming multi-agent systems in Erlang, in: AI*IA/TABOO Joint Workshop on Objects and Agents, WOA 2003, Villasimius, CA, Italy, 2003.
[24] A. Di Stefano, C. Santoro, On the use of Erlang as a promising language to develop agent systems, in: AI*IA/TABOO Joint Workshop on Objects and Agents, WOA 2004, Turin, Italy, 2004.
[25] A. Di Stefano, C. Santoro, Designing collaborative agents with eXAT, in: ACEC 2004 Workshop at WETICE 2004, Modena, Italy, 2004.
[26] A.D. Stefano, F. Gangemi, C. Santoro, ERESYE: artificial intelligence in Erlang programs, in: ERLANG '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, ACM Press, New York, NY, USA, 2005, pp. 62–71, http://doi.acm.org/10.1145/1088361.1088373.
[27] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos, Tropos: an agent-oriented software development methodology, Auton. Agents Multi-Agent Syst. 8 (3) (2004) 203–236, http://dx.doi.org/10.1023/B:AGNT.0000018806.20944.ef.
[28] L. Padgham, M. Winikoff, Developing Intelligent Agent Systems: A Practical Guide, John Wiley & Sons, Chichester, 2004.
[29] J. Pavón, J. Gómez-Sanz, R. Fuentes, The ingenias methodology and tools, in: B. Henderson-Sellers, P. Giorgini (Eds.), Agent-Oriented Methodologies, Idea Group Publishing, 2005, pp. 236–276, Ch. IX.
[30] M. Cossentino, From requirements to code with the Passi methodology, in: B. Henderson-Sellers, P. Giorgini (Eds.), Agent-Oriented Methodologies, Idea Group Publishing, 2005.
[31] A. Van Breemen, K. Crucq, B. Kröse, M. Nuttin, J. Porta, E. Demeester, A user-interface robot for ambient intelligent environments, in: Proceedings of the 1st International Workshop on Advances in Service Robotics (ASER), Bardolino, 2003.
[32] S. Gottifredi, M. Tucat, D. Corbatta, A.J. García, G.R. Simari, A BDI architecture for high level robot deliberation, in: XIV Congreso Argentino, de Ciencias de la Computación, 2008.
[33] S. Rockel, D. Klimentjew, J. Zhang, A multi-robot platform for mobile robots – a novel evaluation and development approach with multi-agent technology, in: Proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), University of Hamburg, Hamburg, Germany, 2012.
[34] L. de Silva, F. Meneguzzi, D. Sanderson, J.C. Chaplin, O.J. Bakker, N. Antzoulatos, S. Ratchev, Interfacing belief-desire-intention agent systems with geometric reasoning for robotics and manufacturing, in: Service Orientation in Holonic and Multi-Agent Manufacturing, Springer, 2016, pp. 179–188.
[35] A.S. Jensen, Implementing Lego agents using Jason, http://arxiv.org/pdf/1010.0150.pdf, 2010.
[36] K. Kravari, N. Bassiliades, A survey of agent platforms, J. Artif. Soc. Soc. Simul. 18 (1) (2015) 11.
[37] T. Semwal, M. Bode, V. Singh, S.S. Jha, S.B. Nair, Tartarus: a multi-agent platform for integrating cyber-physical systems and robots, in: Proceedings of the 2015 Conference on Advances in Robotics, ACM, 2015, p. 20.
[38] A. Soriano, E.J. Bernabeu, A. Valera, M. Vallés, Multi-agent systems platform for mobile robots collision avoidance, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2013, pp. 320–323.
[39] A. Di Stefano, C. Santoro, A3m: an agent architecture for automated manufacturing, Softw. Pract. Exp. 39 (2) (2009) 137–162, http://dx.doi.org/10.1002/spe.894.
[40] R.B. Rusu, L. Miclea, S. Enyedi, Robotux–a multi-agent robot based security system, in: Proceedings of IEEE TTTC Automation, Quality & Testing, Robotics International Conference, Cluj-Napoca, Romania, May 13–15, 2004, AQTRJ, 2004.
[41] M.P. Georgeff, A.L. Lansky, Reactive reasoning and planning, in: AAAI, vol. 87, 1987, pp. 677–682.
[42] F.F. Ingrand, M.P. Georgeff, A.S. Rao, An architecture for real-time reasoning and system control, IEEE Expert 7 (6) (1992) 34–44.
[43] K.L. Myers, A procedural knowledge approach to task-level control, in: Proceedings of the Third International Conference on AI Planning Systems, 1996.
[44] K.L. Myers, User Guide for the Procedural Reasoning System, Technical report, SRI International, Menlo Park, CA, 1997.

[45] M. d'Inverno, D. Kinny, M. Luck, M. Wooldridge, A formal specification of dMARS, in: Intelligent Agents IV, Agent Theories, Architectures, and Languages, Springer, 1998, pp. 155–176.
[46] P. Busetta, N. Howden, R. Rönnquist, A. Hodgson, Structuring BDI agents in functional clusters, in: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), ATAL '99, Springer-Verlag, London, UK, 2000, pp. 277–289.
[47] Jack – an agent infrastructure for providing the decision-making capability required for autonomous systems, AOS Group, http://www.aosgrp.com/downloads/JACK_WhitePaper_US.pdf.
[48] R.H. Bordini, J.F. Hübner, BDI agent programming in AgentSpeak using Jason, in: F. Toni, P. Torroni (Eds.), Computational Logic in Multi-Agent Systems, in: Lecture Notes in Computer Science, vol. 3900, Springer, Berlin/Heidelberg, 2006, pp. 143–164.
[49] R.H. Bordini, J.F. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak Using Jason, Wiley, 2007.
[50] O. Boissier, R.H. Bordini, J.F. Hübner, A. Ricci, A. Santi, Multi-agent oriented programming with JaCaMo, Sci. Comput. Program. 78 (6) (2013) 747–761.
[51] A. Mordenti, Programming robots with an agent-oriented BDI-based control architecture: explorations using the JaCa and Webots platforms, http://amslaurea.unibo.it/4803/1/mordenti_andrea_tesi.pdf, 2012.
[52] A. Pokahr, L. Braubach, W. Lamersdorf, Jadex: a BDI reasoning engine, in: Multi-Agent Programming, Springer, 2005, pp. 149–174.
[53] L. Braubach, A. Pokahr, D. Moldt, W. Lamersdorf, Goal representation for BDI agent systems, in: Programming Multi-Agent Systems, Springer, 2005, pp. 44–65.
[54] V. Nicosia, C. Santoro, Experiences from using Erlang for autonomous robots, in: Proc. of 12th International Erlang User Conference, EUC'2006, Stockholm, Sweden, 2006.
[55] J. Orkin, Applying goal-oriented action planning to games, in: AI Game Programming Wisdom, vol. 2, 2004, pp. 217–227.
[56] A. Klingenberg, Prototypische Entwicklung eines emotionalen Agenten auf der Basis des Goal Oriented Action Plannings, 2010.
[57] The pygame project, pyGOAP, available at http://www.pygame.org/project-pyGOAP-1408-.html, 2013.
[58] RGOAP project, available at https://github.com/felix-kolbe/executive_rgoap.
[59] F. Kolbe, Goal oriented task planning for autonomous service robots, Master's thesis, Hamburg University of Applied Science, 2013, available at http://edoc.sub.uni-hamburg.de/haw/volltexte/2014/2537/pdf/MA_Kolbe.pdf.
[60] F. Messina, G. Pappalardo, C. Santoro, A goal-centric framework for behaviour programming in autonomous robotic systems, in: 2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA), IEEE, 2014, pp. 1–6.
[61] A. Champandard, Behavior trees for next-gen game ai, in: Game Developers Conference, Audio Lecture, 2007.
[62] A. Johansson, P. Dell'Acqua, Emotional behavior trees, in: 2012 IEEE Conference on Computational Intelligence and Games (CIG), IEEE, 2012, pp. 355–362.
[63] A. Marzinotto, M. Colledanchise, C. Smith, P. Ögren, Towards a unified behavior trees framework for robot control, in: 2014 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2014, pp. 5420–5427.
[64] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, Ros: an open-source robot operating system, in: ICRA Workshop on Open Source Software, Kobe, Japan, vol. 3, 2009, p. 5.
[65] L. Fichera, D. Marletta, V. Nicosia, C. Santoro, A methodology to extend imperative languages with agentspeak declarative constructs, in: WOA, 2010.
[66] L. Fichera, D. Marletta, V. Nicosia, C. Santoro, Flexible robot strategy design using belief-desire-intention model, in: Research and Education in Robotics – EUROBOT 2010, Springer, 2011, pp. 57–71.
[67] G. Van Rossum, F.L. Drake, Python Language Reference Manual, Network Theory Ltd., 2011.
[68] Python official documentation, https://docs.python.org/, 2016.
[69] Á.F. Díaz, C.B. Earle, L.-Å Fredlund, eJason: an implementation of Jason in Erlang, in: M. Dastani, J.F. Hübner, B. Logan (Eds.), Programming Multi-Agent Systems, 10th International Workshop, ProMAS 2012, Valencia, Spain, June 5, 2012, Springer, Berlin/Heidelberg, 2013, pp. 1–16, revised selected papers.
[70] Jack intelligent agents agent manual, AOS group, http://www.aosgrp.com/documentation/jack/Agent_Manual_WEB/index.html.
[71] J. Gosling, B. Joy, G.L. Steele, G. Bracha, A. Buckley, The Java Language Specification, Pearson Education, 2014.
[72] G. Fortino, F. Rango, W. Russo, C. Santoro, Translation of statechart agents into a BDI framework for mas engineering, Eng. Appl. Artif. Intell. 41 (2015) 287–297.