Towards a Real-Time BDI Model for ROS 2

Francesco Alzetta
University of Trento
Trento, Italy
francesco.alzetta@unitn.it

Paolo Giorgini
University of Trento
Trento, Italy
paolo.giorgini@unitn.it

***Abstract*—In the race for automation, electronic devices are required to become more and more intelligent in order to make the correct choices in unforeseen situations without any need of human intervention. AI proposes basically two different approaches: machine learning algorithms and multi-agent systems. While the former perform very well when dealing with single, independent, computing units, multi-agent systems are more suitable in case of different components interacting with one another. In this paper, we propose a real-time multi-agent approach to improve practical reasoning, integrating the Belief-Desire-Intention model into one of the most popular robotics framework, ROS 2.**

***Index Terms*—BDI model, ROS 2, Multi-Agent Systems, Soft Real-time, Robotics framework, Smart devices**

I. Introduction

Robots and devices involved in complex environments generally require both high level, deliberative, capabilities (such as reasoning, knowledge representation, planning, communication) and low level, reactive, primitives (such as sensor management, obstacle avoidance, navigation). While nowadays the reactive capabilities are quite satisfyingly achieved, with devices able to reliably sense the environment and promptly react to stimuli, the deliberative part still presents a number of open problems.

Reasoning and planning, for instance, are activities that usually consume time and resources, making them unsuitable for real-world scenarios involving devices with real-time constraints.

Furthermore, for many years robotics developers struggled with a large variety of open source and proprietary standards, which forced them to find ad hoc solutions to make possible the interaction and communication between devices sold by different vendors. Robot Operative System (ROS) [1] solves this problem, giving the developers an across-the-board tool able to deploy the same code on devices of different vendors.

ROS 2 overcomes some of the ROS limitations [2], such as the possibility to write real-time nodes when using a proper Real-Time Operating System (RTOS) and the implementation of a distributed structure for discovery and interaction between nodes.

In this paper, we propose a new approach that allows integrating the notion of *agent* into devices running ROS 2. We develop a model of a real-time agent, implementing a BDI architecture [3] natively by using the core functionality of the middleware. The choices of the agent, then, are influenced not only by its internal status and by the stimuli coming from the environment, but also by time restrictions. Our proposal represents a first step towards implementing in ROS 2 the concept of agents whose practical reasoning is bounded for real-time performances.

In Section II we briefly cover the background knowledge needed to understand the technical part of the work. In Section III we show the architecture of our system, discussing the most interesting details about how we implemented it in Section IV. We conclude with an overview of the related work, followed by a discussion about the future improvements and the contributions of this work.

II. Background

Despite its name, ROS 2 is not an operative systems, but it can rather be seen as a distributed framework of processes (or *nodes*) that enables executables to be individually designed and loosely coupled at run-time.

As mentioned in the introduction, there are many improvements made in ROS 2, concerning in particular real-time compliance and system scalability, which led us to choose it instead of its widely adopted first version. Moreover, ROS 2 has several desirable properties that are missing in other robotics frameworks taken into consideration such as OROCOS [4], YARP [5] or CARMEN [6], namely:

- **Modularity:** This characteristic allows a developer to decide which parts of the general implementation take from the packages shared by other developers and which implement by himself instead. In our case, for instance, we focused mainly on the development of the BDI system, leaving the management of other tasks such as locomotion or sensing to modules designed and implemented by other developers.
- **Compatibility:** Being ROS designed to be as thin as possible, the code written for it can be easily integrated inside other robot software frameworks. Some ROS code has been already integrated with OpenRAVE[7], Player[8] and OROCOS.
- **Language independence:** Having the possibility of writing different parts of the code with different languages (C++, Python, Lisp, Java are some of the supported languages) allows the developer to exploit the advantages of each language and their libraries.
- **Wide adoption:** ROS is a very popular framework among the robotics developers and manufacturers so, by choos-

ing it, we are widening the possible application fields of our work.

The main concepts of ROS 2 are: *packages*, *nodes*, *messages*, *topics*, and *services*.

- **Packages** are collections of nodes, datasets, configuration files and anything else that logically constitutes a useful module, where module is intended as an easy-to-consume piece of software that can be nimbly reused by other users.
- **Nodes** are processes responsible for performing computations. They can be seen as entities that can execute code and communicate with each other.
- **Messages** are simple data structures that comprise typed fields. Standard primitive types (integer, floating point, boolean, string) are supported, as long as arbitrarily nested structures and arrays. Messages are passed between nodes when they have to share information.
- **Topics** can be seen as containers in which the publishing nodes send out messages that will be gathered by the subscribing nodes. For each topic, there may be multiple concurrent publishers and subscribers, and each node may publish and/or subscribe to multiple topics. This publish-subscribe pattern allows the system to be more scalable, since new nodes can be added and start publishing and/or subscribing to already existing topics in a totally transparent way to all the other nodes, which do not need to be aware of the newly joined nodes.
- **Services** are used when the designer of the system wants request/reply interactions between two nodes, instead of the many-to-many mechanism of the publish-subscribe pattern. They are defined by a pair of message structures, one defining the request a client can send towards the provider of the service, and one representing the structure of the reply the provider should send back to the client. Services, then, should be used when a one-to-one communication is needed.

The tangible part of our work, then, is represented by a package containing design skeletons and examples of implementation of some nodes which are interacting through the exchange of messages published on topics or sent via a service.

III. Architecture

In this section, we show the multi-layered architecture we propose to integrate BDI agents in ROS 2.

The general architecture

In Fig. 1 we show how our implementation (in orange) wedges in the general ROS 2 architecture.

Starting from the bottom, the proposed architecture can be deployed without restrictions in any operative system running ROS 2, even though an RTOS is necessary when running systems that need hard real-time behavior. The Data Distribution Service (DDS) implementation provides a publish-subscribe transport that allows any two DDS programs to communicate without the need for a central coordinator. This

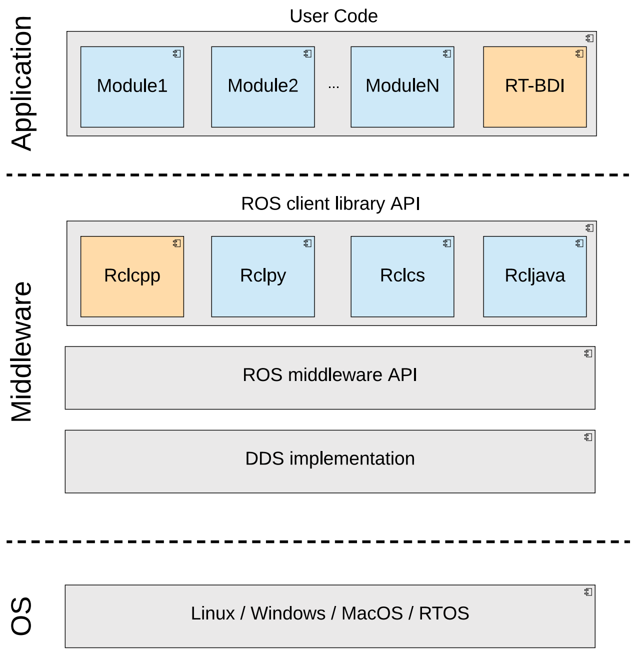

Fig. 1: The software architecture.

layer extremely simplifies the job to the developer, since it avoids him the task of setting up a connection between two nodes to exchange information. The ROS middleware API, instead, is responsible for providing support to multiple DDS implementations despite the fact that each of them differs slightly in their exact API. This interface, then, allows using a general publish-subscribe system in the upper layers without worrying about the specific DDS implementation used. The ROS client library API layer makes ROS concepts easy to use and to access via code. It allows the programmer to implement its own nodes in the desired language (obviously provided that a client library for ROS exists for that language), being sure that it will be fully compatible with all the other nodes in the system, even those written in a totally different language. The upper layer is where the code written by the designer of a package is located. All the nodes run by the device reside here, hence this is the level at which our work has focused most. In the following, we present the architecture of our agent, showing how the ROS 2 nodes make use of the BDI model.

The real-time BDI architecture

We designed our BDI model to explicitly consider timing constraints in the actions of the agents and in the interactions between them. This is particularly helpful when assuming that agents have different ways to reach their goals, so that timing restrictions play an important role in deciding which actions to perform in order to achieve the desired result.

In the real-time domain, there is a clear distinction between *soft* and *hard* real-time. These two terms do not differ in how the real-time is guaranteed, but rather in the extent of the damage caused to the system in case the real-time constraints are not met. In a soft real-time agent, missing a deadline may prevent the data to be processed or the behavior of the agent from being accurate, but it will not be considered as a catastrophic failure by the system. Conversely, a hard real-time system will treat as a failure the non-fulfillment of its temporal restricted responsibilities.

Since our work operates at a high-level layer, being almost completely transparent to those low-level components responsible for achieving hard real-time constraints, it can only guarantee soft real-time constraints. Missing a deadline, then, should be tolerated by the system.

In our model, the designer of the MAS will define a *deadline* for desires, namely the time the agent has to accomplish that goal, while for plans it represents the maximum time required to complete a plan.

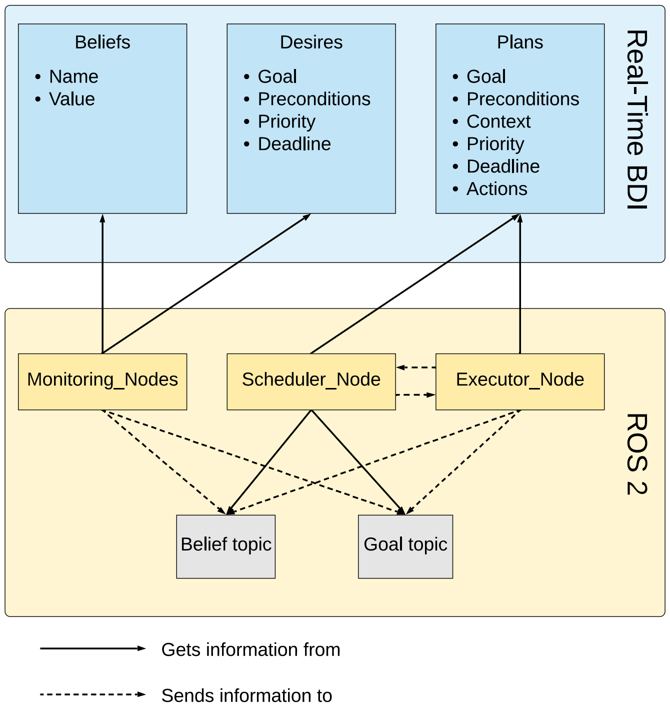

Fig. 2: The agent architecture.

A real-time agent is represented as a tuple defined by the name of the agent, its belief-set, its desire-set and the set of all possible applicable plans. These are pre-defined, and they represent what an agent could know about the world, which are the goals it could achieve, and how it can interact with the surrounding environment to accomplish that goal. Currently, a plan is a sequence of actions defined by the designer but, in future work, we will consider the development of an automated planner in order to make more flexible agents, which will be able to autonomously build a plan.

In Fig. 2 we illustrate the relationships between the elements constituting an agent, highlighting how the ROS 2 part interacts with the real-time BDI model.

Beliefs

The elements of the belief-set are defined as couples composed by a string, which identifies the belief, and a value assigned to that belief, which could be of any type supported by ROS 2 messages. In the following, the BNF grammar of a belief.

```
<belief>   ::= <name> <term_exp>
<name>     ::= string
<term_exp> ::= boolean | string | int | float
```

Desires

The elements of the desire-set are expressed by a tuple: the goal that has to be achieved, represented by the desired belief, a set of preconditions that will trigger the desire when all of them are satisfied, a variable stating which is the priority of that goal and a variable indicating the deadline, that is the time the agent has to complete the intention selected to achieve the goal.

```
<desire>  ::= <belief> <precond> <priority> <deadline>
<precond> ::= ε | <precond> <belief>
<priority>::= int
<deadline>::= float
```

In future work, the last two parameters should be inferred directly by the agent, that should be able to autonomously decide which goal should have priority over the others, and how much time each desire should take in order to be satisfied. In this phase, predetermine such values is a task left to the designer.

Plans

The plan-set contains all the possible plans an agent can execute. The scheduler will choose a plan on the basis of the goal it wants to achieve, on its current beliefs, on its priority and on the executing time being available.

```
<plan>     ::= <name> <desire> <precond> <context>
               <priority> <deadline> <body>
<name>     ::= string
<precond>  ::= ε | <precond> <belief>
<context>  ::= ε | <context> <belief>
<priority> ::= int
<deadline> ::= float
```

Indeed, when specifying a plan, the designer should explicit, besides the *body* of the plan (i.e. the actions to perform), also the priority of that plan (in order to make the scheduler choose rapidly among a number of possible plans) and the maximum time required by the plan to complete.

We decided to split the concept of *current beliefs* into two different data structures, preconditions and context, since the conditions necessary for the activation of a plan could be different from the ones that must hold during its execution. For instance, a drone could require to be fully charged as the only precondition to start a journey, but requiring the wind speed to stay below 10 km/h to continue its traveling and not land immediately. Preconditions are those beliefs that must hold at the very start of the plan selection, so the necessary conditions for the scheduler to consider the plan an applicable plan in the current situation. The context, instead, is the set of beliefs that must hold for the entire execution of the plan, causing the failure of the plan otherwise. This is particularly useful in highly dynamic environments where important beliefs are updated frequently. Obviously, not all the plans with the same goal must have the same preconditions and context.

The agent's nodes, running in Ros 2, use these structures to instantiate at run-time the actual beliefs, goals, and intentions of the agent.

The **monitoring nodes** (i.e. those nodes responsible for sensing the environment and monitoring the internal state of the agent) take from the real-time BDI model the representation of those beliefs and desires that could belong to the agent, and instantiate at run-time a message corresponding to the actual belief/desire when perceived/generated. This message, then, will be published to the appropriate topic.

The **scheduler node**, which is the reasoner of the agent, is continuously listening on belief and goal topics, updating the belief-set and the desire-set whenever a message on these topics is published. Furthermore, since such an update could be meaningful for the failure or instantiation of a plan, or for the activation of a pending one, the scheduler will check if a reschedule is necessary every time a node publishes to those topics.

In case a new schedule has been processed, this will be sent to the executor node, which is the node that actually executes step by step the actions of the currently selected plan. An executor node is necessary because executing a plan is a blocking operation, and leaving this task to the scheduler would have meant preventing any rescheduling until the current plan either fails or succeeds, causing the system to be very unresponsive.

The **executor node**, during the execution of a plan, will send messages to the belief and goal topic to make all the nodes subscribed to them know about the changes in agent's belief or the completion of a goal. This is important because some beliefs cannot be generated by the monitoring nodes, but have to be inferred from the actions of the agent. For instance, a robot vacuum could not rely on sensors to know if it has scoured the entire room, but this information can be provided from the executor node, that will publish the belief *Room_cleaned, true* when a plan for cleaning the room has been completed, implying that the room has also been entirely scoured.

In the next section, we discuss more in depth how we implemented the architecture in ROS 2.

IV. Implementation

As briefly introduced previously, our package strongly relies upon the advantages provided by the publish-subscribe system implemented in ROS 2.

The reason why we decided to make the monitoring nodes and executor use topics instead of services, preferring a many-to-many communication even though a one-to-one could seem more straightforward, is scalability. With this solution, in fact, the designer should not take care of the communication between a newly added node and other nodes which can potentially be interested in the info the node is meant to share, but it is sufficient that the new node publishes the message to the proper topic.

The main listener on belief and goal topics is the scheduler that, by subscribing to them, is constantly updated on the state of the device. This node, every time either beliefs or desires are updated, reschedules the intentions of the agent on the basis of the priorities and deadlines of its current desires, if necessary. This choice is made by exploring the set of plans, searching for all the applicable ones, and scheduling a sequence of plans which can guarantee at least the achievement of the most critical goals (i.e. the ones with the highest priority).

Once the scheduler has decided the order in which the plans should be executed, it communicates the sequence to the executor via a service, as it is the only node interested in this information. The main task of the executor is to control the execution and to publish messages to the belief topic, in the case where an action performed by the agent changes any of its states, or to the goal topic when an intention finishes, achieving the goal. If the plan, instead, finishes due to a failure, there can be two reasons: the first one involves the executor encountering an error during the execution (e.g. an exception is raised). In this case, the node sends a message to the scheduler node via a service, asking for a rescheduling. The second reason is that one condition of the context does not hold anymore. In this case, the executor should do nothing, as the scheduler will be autonomously aware of the fact, starting immediately a rescheduling.

We now illustrate the most relevant parts of the implementation and the solutions we found to overcome some of the limits of ROS 2.

Message structure

ROS 2 does not allow to use generic types for message fields, hence there is no way to use the same message to represent, for instance, both a boolean-typed and a string-typed belief. To cope with this limitation, we took into consideration two different solutions: the first one consists in providing eight different types of messages, representing the four standard primitive types (boolean, string, int, and float) for both beliefs and desires, while the second one involves only two types of messages, one for beliefs and one for desires, having a field for each primitive type.

Despite the last choice would allow a cleaner code, it forces the developer to find a way to make the nodes understand which is the intended value type among all the fields.

We implemented four different ROS 2 messages describing a belief (containing the name of the belief and its corresponding value) and four different ROS 2 messages describing a desire (containing the goal's name, value, priority, and deadline).

Belief and desire representation

Beliefs and desires can be represented by messages when they are simply treated as information that should be exchanged between nodes, but when nodes need to use that information; for instance, to check if they have a suitable plan to deal with an incoming desire, nodes have to be encapsulated in a more standard structure to be easily handled. For this reason, when the scheduler node acquires the information that a new belief or desire has been published on the proper topic, it instantiates a new Belief or Goal object, passing into the constructor the fields contained in the message. Then, this new object will be added to the belief-set if being a belief, or will try to activate a new plan if being a goal.

Obviously, since we have eight different messages, we also need eight different classes to encapsulate every type of possible belief and desire.

Nodes

As introduced in Section III, we have different nodes with different purposes. The monitoring nodes give information about the perception the agent has of the environment and its internal state, instantiating also the goals when their preconditions are met. In a real-case scenario, the majority of such nodes are implemented by the vendor of the device running ROS 2, and the only job left to the designer of the agent is to forward the information coming from these nodes to the appropriate topic. The scheduler is the main consumer of this information.

In the following, Algorithm 1 describes the main function to reschedule an intention.

Algorithm 1

```
1:  procedure RESCHEDULEINTENTION(Goal g)
2:      possiblePlans ← empty array of possible plans
3:      chosenPlan ← the plan to be executed
4:      for each plan $p$ in plan-set do
5:          if $p.verifyGoal(g)$ & $p.verifyPrec(desire\text{-}set)$ then
6:              add $p$ to possiblePlans
7:      sortBySuitability(possiblePlans)
8:      for each plan $p$ in possiblePlans do
9:          if checkIfSchedulable(p) then
10:             chosenPlan ← $p$
11:             break
12:     sendToExecutor(chosenPlan)
```

It is composed of four steps: first, it collects all the possible plans by selecting those designed to reach the desired goal and whose preconditions are respected (lines 5-7). Then, it sorts them by suitability. In our implementation, the most suitable plan is the one having the greatest priority value being less or equal than the goal's priority value (here we assume that lower is the priority value, more urgent is the desire). This way, we give precedence to those plans that at the same time respect the priority required by the goal and are the less invasive, hence leaving room for more urgent plans that may come later. The third step involves deadlines: the algorithm checks every plan in the sorted order until it finds one whose deadline fits in the current schedule (lines 9-12). In case there is no way to insert that plan into the current schedule, the scheduler checks if the desire it is taking into consideration has a higher priority then any of the desires for which a plan has been previously added to the schedule. If that is the case, that plan will be deleted from the scheduling queue and the third step is repeated, otherwise the considered desire is added to the pending ones, waiting for a new reschedule. We preferred to not include this case in our pseudocode for clarity. Finally, the new schedule is sent to the executor, which will manage the execution of the new intention.

Plan structure

To ensure that the scheduler will be able to process all the plans made available by the designer, we decided to force every plan to derive from a superclass called Plan. This decision brings at least two advantages: the first one is that, since every plan has the same basic declarations, we guarantee that all the plans have the same structure and the same fundamental functions. Secondly, this allowed us to have collections of plans, simplifying a lot the data structures in both the scheduler and executor nodes.

Scheduler and executor, then, are guaranteed that the three functions declared in the superclass will be defined in every instantiated plan. We illustrate these functions by showing how we designed a robot vacuum's plan for cleaning a room.

Algorithm 2 defines how the scheduler should check if the plan is suitable to achieve the requested desire. In our example, the scheduler checks if the instantiated goal has the same name of the plan's goal (e.g. *clean_room*) and the same value (e.g. *true*).

Algorithm 2

```
1:  procedure VERIFYGOAL(Goal goal)
2:      if $plan.goal.name = goal.name$ then
3:          if $plan.goal.value = goal.value = true$ then
4:              return true
5:      return false
```

Algorithm 3 has to implement the procedures needed to check if the plan is applicable given the agent's set of beliefs. In our case we designed the plan so that the first precondition is *battery_charge, 15*, which means that a minimum of 15 % of remaining charge is necessary for the robot to start this plan (otherwise the function will return false). The second precondition is *room_clean, false*, hence the agent should have the perception (i.e. the belief) that the room is dirty.

Algorithm 3

```
procedure VerifyPreconditions(Belief[] belief-set)
    for each belief $b$ in belief-set do
        if $b.name = plan.precondition[0].name$ then
            if $b.value < plan.precondition[0].value$ then
                return false
    for each belief $b$ in belief-set do
        if $b.name = plan.precondition[1].name$ then
            if $b.value \neq plan.precondition[1].value$ then
                return false
    return true
```

Algorithm 4 defines the actions the agent should perform in case the scheduler chooses that plan as the next plan to be executed.

Algorithm 4

```
procedure activatePlan
    $goto(room)$
    $startVacuum$
    while $room\_clean = false$ do
        $roam(room)$
    $stopVacuum$
```

The designer of a plan, besides implementing these three functions, has to instantiate in the class constructor the parameters declared in the superclass, namely: the plan's name, the goal that it achieves, preconditions, context, priority, and deadline.

V. Related work and future work

The source code and an implementation example are available at https://github.com/ElDivinCodino/ROS2BDI. In such an example, we simulated the behavior of a robot vacuum having three different sensors that check the cleanliness of the rooms, the charge of the battery, and the filling level of the dirt tank, and being able to activate different plans depending on the situation. This implementation, despite a scheduling algorithm being still raw, shows that the robot is able to autonomously take smart decisions, planning mid-term strategies in order to adapt to the priorities given by the different desires that originate during the simulation. Further experiments involving a simulated MAS are planned, so as to have a more complex environment that allows us to formally compare our work with other state-of-the-art frameworks, as JADE [9], Jason [10] or JACK [11].

JADE, a framework written in Java, simplifies the development of intelligent multi-agent systems by offering to the developer desirable features, such as FIPA-compliant specifications, a distributed platform (agents can be split on different hosts), parallel task execution and a GUI to manage several agents and agent platforms. The JACK framework exploits the Java Virtual Machine to build multi-agent architectures. It implements the BDI model by extending the Java language, adding concepts such as *agent*, *plan*, and *event* as first-class components of the language. Although JADE and JACK are very powerful tools, extending an already existing language requires constant maintenance of the framework. Indeed, JACK's syntax supports Java versions until J2SE 1.4, hence important features such as annotations and generics are not available. Due to the structure of ROS 2, we do not face this problem, since it is continuously maintained by the ROS community, which deals with the changes in the language by updating the ROS client library API layer. We took inspiration from JACK for the design of our plans, which describe the exact sequence of actions an agent should perform, in order to reach the goal, when a given event occurs. However, in future, we will also consider implementing a planner that allows the agent to autonomously build a plan, similarly to what currently do solvers based on STRIPS [12] and PDDL [13] languages. This will greatly improve the adaptability of the agent, making it able to find a suitable plan also in unforeseen situations. Another relevant improvement, to make the design of our agent easier, concerns a design tool similar to the one developed for JACK. Indeed, the main problem with ROS 2 is about its learning curve, as while once reached an intermediate level of knowledge ROS 2 is pretty easy to manage, the first steps can be problematic, especially for a developer which does not have a strong C++ background. Such a tool would provide the visual representation of the various components and of the links between them, creating and editing the appropriate design skeletons, i.e. well-formed and already linked files whose implementation can be developed further at any time.

During the design of our BDI architecture, we followed the real-time approach proposed in the ARTS architecture [14]. This work is particularly interesting because it introduces in a BDI architecture the concepts of deadline and priority for goals and plans, scheduling intentions so as to make the highest priority intentions achieved by their deadlines. This comes together with a *deadline monotonic intention scheduling algorithm* which, although it does not output the optimal schedule, gives precedence to the most urgent intentions. We designed our scheduler to have the same behavior. An important improvement to the decision-making process of the agent concerns adding a dynamic inference of the priority and the deadline of a desire. Indeed, while now the task of assigning such values to each possible desire is left to the designer, there could be situations where these values may change on the basis of the situation, or simply the designer could not know a priori how much time a plan will need in order to complete.

The idea of having a process dedicated to the management of the execution of intentions is inspired by [14] and [15]. These architectures, however, demand to one single process (called, respectively, *executor* and *interpreter*) both the activities of selecting and executing a plan, while we decided to split these tasks into two different processes for the reasons explained in Section III.

Concerning strictly the robotics field, the BDI model is often applied in very specific contexts such as robot competitions [16] and social interactions [17], [18]. However, few general-purpose BDI implementations, such as the previously mentioned CogniTAO [19], exist. However, while CogniTAO implements the BDI model in the ROS architecture by creating an ad-hoc structure, delivering the execution of the agent paradigm to a unit (called TAO machine) logically separated from the other ROS-related components, in our work the BDI model is completely integrated with the ROS main concepts, since it strongly relies upon them. Furthermore, real-time constraints are not considered in CogniTAO, and there is no scheduler that decides the order of the execution of the tasks.

The problem of achieving agent cooperation has been treated in an initial phase of our study, but the actual implementation of a distributed computing algorithm that allows splitting tasks among agents has been left as a further improvement.

VI. Conclusions

We have proposed an approach to enhance the decision-making process of robotics devices by exploiting the core mechanisms of the ROS 2 middleware. Our work fully integrates the BDI model in the ROS 2 architecture, in the sense that it does not need the support of ad-hoc structures or the execution of external software. Moreover, it provides support for real-time, allowing the agent to take into consideration time when generating its intentions. We acknowledge that the current implementation merely shows that this approach is possible, but we planned to test it in more complex scenarios. Lastly, we have planned future work that facilitates the user experience and the design of a MAS, and improvements on the scheduler and agent capabilities.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[2] V. A. Hax, N. L. Duarte Filho, S. S. da Costa Botelho, and O. M. Mendizabal, “Ros as a middleware to internet of things,” *Journal of Applied Computing Research*, vol. 2, no. 2, pp. 91–97, 2013.

[9] F. Bellifemine, A. Poggi, and G. Rimassa, “Jade–a fipa-compliant agent framework,” in *Proceedings of PAAM*, vol. 99, no. 97-108. London, 1999, p. 33.

[3] A. S. Rao and M. P. Georgeff, “Modeling rational agents within a bdi-architecture.” *KR*, vol. 91, pp. 473–484, 1991.

[4] H. Bruyninckx, “Open robot control software: the orocos project,” in *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, vol. 3. IEEE, 2001, pp. 2523–2528.

[5] G. Metta, P. Fitzpatrick, and L. Natale, “Yarp: yet another robot platform,” *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.

[6] M. Montemerlo, N. Roy, and S. Thrun, “Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit,” in *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2003, pp. 2436–2441.

[7] R. Diankov and J. Kuffner, “Openrave: A planning architecture for autonomous robotics,” *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, vol. 79, 2008.

[8] B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems,” in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.

[10] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, vol. 8.

[11] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas, “Jack intelligent agents-components for intelligent agents in java,” *AgentLink News Letter*, vol. 2, no. 1, pp. 2–5, 1999.

[12] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[13] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “Pddl-the planning domain definition language,” 1998.

[14] K. Vikhorev, N. Alechina, and B. Logan, “The arts real-time agent architecture,” in *International Workshop on Languages, Methodologies and Development Tools for Multi-Agent Systems*. Springer, 2009, pp. 1–15.

[15] D. Morley and K. Myers, “The spark agent framework,” in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*. IEEE Computer Society, 2004, pp. 714–721.

[16] S. Gottifredi, M. Tucat, D. Corbatta, A. J. García, and G. R. Simari, “A bdi architecture for high level robot deliberation,” in *XIV Congreso Argentino de Ciencias de la Computación*, 2008.

[17] A. Van Breemen, K. Crucq, B. Kröse, M. Nuttin, J. Porta, and E. Demeester, “A user-interface robot for ambient intelligent environments,” in *Proc. of the 1st Int. Workshop on Advances in Service Robotics,(ASER)*. Citeseer, 2003, pp. 132–139.

[18] B. R. Duffy, R. Collier, G. M. O’Hare, C. Rooney, and R. O’Donoghue, “Social robotics: Reality and virtuality in agent-based robotics,” in *Bar-Ilan Symposium on the Foundations of Artificial Intelligence: Bridging Theory and Practice (BISFAI)*, 1999.

[19] Cogniteam, “Cognitao (bdi),” 2014. [Online]. Available: http://wiki.ros.org/decision_making/Tutorials/CogniTAO