Deliberative Agents in Dynamic Environments, Using Jason and NARS

valeria seidita [1], francesco lanza [2], Patrick Hammer [2], Antonio Chella [2], and Pei Wang [2]

[1]University of Palermo
[2]Affiliation not available

October 30, 2023

Abstract

This work explore the possibility to combine the Jason reasoning cycle with a Non-Axiomatic Reasoning System (NARS) to develop multi-agent systems that are able to reason, deliberate and plan when information about plans to be executed and goals to be pursued is missing or incomplete. The contribution of this work is a method for BDI agents to create high-level plans using an AGI (Artificial General Intelligence) system based on non-axiomatic logic.

Deliberative Agents in Dynamic Environments, Using Jason and NARS

Francesco Lanza[1], Valeria Seidita[1], Patrick Hammer[2], Pei Wang[2], and Antonio Chella[1]
[1] Dipartimento di Ingegneria, Università degli Studi di Palermo, 90128, Viale delle Scienze, Palermo, Italy
[2]Department of Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, PA 19122

Complex software systems operating in dynamic environments require a high degree of adaptability and deliberative skills, similar to humans. In recent years, multiagent systems have been shown to be suitable for solving problems in complex domains. To effectively solve and manage complexity, we need agents that are able to find plans to pursue goals by interacting with the workspace without the need for programmers or designers to intervene. We need agents that are able to plan and reason independently during the execution phase. We propose to use the *Belief- Desire-Intention* agent paradigm and its implementation by the reasoning cycle *Jason* together with a Non-Axiomatic Reasoning System (NARS). The result is a reasoning system that allows an agent to choose the most useful plan for a given situation, while withdrawing unsuccessful plans and learning new plans on the fly.

***Index Terms*—Agent Oriented Programming, Reasoning System, JASON, Adaptive behaviour, NARS**

I. Introduction

In complex domains where a high degree of adaptability of software systems is required, one of the main challenges is to equip the system with the ability to plan during its execution. During execution, the system interacts with users and the environment, which often continuously change as a result of the interaction. Some examples are IoT systems, Human-Robot Interaction, social or economic systems. Problems in these scenarios can be solved by software systems that are able to adapt to changing situations and plan work even in the complete or partial absence of all useful input data.

These aspects cannot be faced and solved at design time. Developers cannot identify and implement all possible situations where a high degree of autonomy is required. At best, they can identify multiple if-then conditions and consider a range of possible alternatives in system behavior.

The more dynamic or uncertain the environment, the more developers need to implement mechanisms that allow the system to autonomously revise a plan or sometimes select or create a new one (adaptive and contextually changing behavior).

An efficient way to solve the above problem is to use the agent-oriented paradigm. An agent is an autonomous entity that is able to respond to stimuli from the environment and proactively work towards a specific goal [1]. Specifically, the BDI agent paradigm [2] involves a deliberative capacity of the agent based on a continuous sense-action loop and existing beliefs. It allows the agent to realize a desire with a plan available in its library. Several technological approaches in the literature describe possible implementations of BDI agents. One of the best known and most efficient is the *Jason* Framework and its reasoning cycle [3], [4]. *Jason* is a powerful tool for implementing planning in uncertain environments. However, its rigid plan-finding procedure and predefined plan library make it unsuitable for the above purposes.

Manuscript received XXX; revised XXXX. Corresponding author: Valeria Seidita (email: valeria.seidita@unipa.it).

The idea we present in this paper is to combine the *Jason* reasoning cycle with a Non-Axiomatic Reasoning System (NARS) [5], [6] to develop multi-agent systems that are able to reason, deliberate and plan when information about plans to be executed and goals to be pursued is missing or incomplete.

Typically, both the *Jason* framework and NARS can be used for planning. *Jason* has a reasoning cycle implemented and coded in an agent that requires very specific and defined inputs to select a plan. NARS is a general purpose reasoning engine that does not have this limitation. NARS allows the construction of plans even in the absence of specific knowledge. It is based on information obtained from observed correlations or structural similarities between existing plans or both. By combining the two systems, we create a more complete system whose core consists of one (or more) *Jason* agents, typically executing the usual reasoning cycle and NARS. NARS is used as an external reasoning system that helps the *Jason* agent change the behaviors defined at design time. Each time the *Jason* agent is unable to select a plan from its plan library, it launches a method that uses NARS. NARS generates a plan that is added to the Jason agent's plan library. NARS operates primarily as part of an intelligent system's brain, taking control only when all other automated brain functions are unsuccessful, while listening to beliefs in the background. NARS processes plans and reorganizes them to create new applicable plans when the system fails to achieve a goal.

Jason is able to plan using first-order logic, while NARS uses non-axiomatic logic. The main difference between these systems is in the way they plan. *Jason* plans activities to pursue a goal using the plan library given at design time by the developer and the set of preconditions. Preconditions are associated with each plan and are used to select the best plan, according to the agent's perception. NARS uses Non-Axiomatic Logic (NAL) inference rules, which are well defined in [6] and allow an artificial agent to plan even under the assumption of insufficient knowledge and resources. By combining these approaches, it is possible to increase the performance of the overall system in terms of finding a new

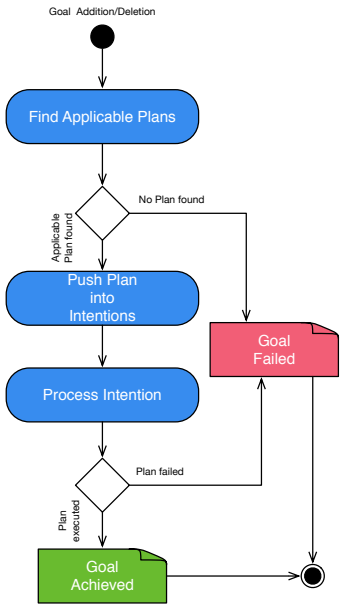

Fig. 1. AgentSpeak control cycle. Redrawn from [9].

strategy that combines these two aspects.

The contribution of this paper is a method for BDI agents to create high-level plans using an AGI (Artificial General Intelligence) system based on non-axiomatic logic. The rest of the paper is structured as follows: Sections II and III show Jason and NARS, their properties and how they handle the selection of plans during the execution phase; Section IV discusses the new reasoning cycle that merges the functions of *Jason* and NARS; Section V shows how plans are created and composed with an example; Section VI illustrates some related works and finally Section VII discusses and draws conclusions.

II. Planning with *Jason* Agents

Jason is a multi-agent framework based on Java and used to create multi-agent systems. *Jason* offers the opportunity to develop applications using the multi-agent approach employing the model proposed by Bratman in [7], also known as the BDI model (*Beliefs, Desires, Intentions*).

The BDI model is inspired by the human behavior and consists of:

- *Beliefs* - representing information that agents have about the environment, called the current context;
- *Desires* - all the possible *states of affairs* that agents want to realize.
- *Intentions* - the *states of affairs* that agents decide to commit according to some available plans. The desires the system has decided to commit to, by executing the associated selected plans to realize them.

Agents defined in *Jason* implement the BDI model through the definition of a set of beliefs or rules, goals and plans written in AgentSpeak(L) [8]. Each agent executes a reasoning cycle starting from perceptions, the result is the execution of an action.

A *Jason* agent presents three main components: the belief base, the agent's goal and the plans. To achieve goals, agent executes plans.

It is worth noting that *Jason* agents are not purely reactive. They compute the result of an action, but they do not terminate. They are designed to stay running and to respond to events by executing plans. This property was fundamental to the realization of the proposed approach and gives agents the ability to plan at runtime. *Jason* agents deliberate actions accordingly with their plan library. The deliberation process of a *Jason* agent is based on practical reasoning. The need to implement this kind of *mentalistic* computer programming underlies BDI agents, and hence *Jason* agents. Practical reasoning is inspired by the human counterpart of practical reasoning, which consists of two distinct activities involving a deliberation phase, something reminiscent of the concept of intentions, and a means-end reasoning, or also, the process that leads the reasoning system to a decision to satisfy the agent's intentions [1]. In *Jason* agents, event-based mechanisms trigger a plan to respond to events that execute plans. In order for an agent to create high-level plans at runtime, this behavior is exploited to invoke another planning system when no other alternative solutions could be drawn.

Goals (hence *Desires*) are in some sense the complement of beliefs. Beliefs represents something that agent believes to be true in the environment. Goals represent the state of the world the agent wants (or commits, in *Jason* terms) to be true. A change in the agent' s beliefs or in the agent's goals triggers the execution of plans (hence *Intentions*). Moreover, a change in the environment causes a change in the agent's beliefs, new opportunities and the possibility to adopt new goals and also drop the existing ones.

A plan is composed of three parts: the triggering event, the context and the body. Its form is:

triggering_event : context <- body

The triggering event is defined by the same word "triggering". It is an event, a part of each plan in the plan library. When the triggering event matches with an event of the environment, which the agent beliefs to be true, then that plan is executed by the agent. The context refers to the concept that agent are reactive but continuously listening of what happens. An agent may have different ways (plans) to achieve a goal. The context allows to check the current situation and decide whether a plan in the library can be activated to reach a goal. The triggering event and the context are the part of the *Jason* agent allowing to handle unknown and changing environment.

Finally, the body is the part of the plan that specifies the set of actions to be taken to achieve the goal. In general, actions aim to change the state of the environment. By the environment we mean the world outside the agent, outside its mind. Jason also allows to consider actions dedicated to the agent's reasoning cycle. This type of actions is called "Internal Action" and simulates an action requested by the reasoner that changes the agent's internal world. Jason provides standard internal actions as well as allows to create new ones. The creation of an internal action serves to extend agents' internal capabilities

and also allows to use legacy code already available, mainly in Java.

A plan that has a *true* context is called *applicable*, the context is validated with first-order logic over preconditions. The precondition is a term that is unified with the content of the namesake belief by the unification process.

A special kind of plan is the *recovery plan*. It can be defined by the programmer to face situations where the context turns out not to be true. It is at the end of the list of plans that may be applicable.

Despite the complexity of the reasoning cycle, the real agent control cycle (Fig. 1) is very simple. For each goal, the agent tries to find an applicable plan in its *Plan Library* . If no plan can be selected, the goal fails, the agent stops the reasoning cycle, and no action is selected. If the agent finds an applicable plan, the agent pushes the plan into the intentions queue and then the reasoning cycle processes the intention. If the intention is processed, the goal is achieved, otherwise the goal has failed.

The agent checks which plan given at design time by a programmer can be applied to the context by evaluating the circumstances of the plan with the context requirements. The list of applicable plans is converted into a queue of intentions, which the reasoning cycle executes as actions step by step through a Round-Robin algorithm. If no plans are marked as applicable, the agent is not able to achieve the goal with the given plan library.

The body is the simplest part of a plan. Each goal has a plan of the same name, which serves as the starting point for the reasoning cycle. The goal is achieved by executing a plan that contains a list of subgoals. Each subgoal (or plan) contains a set of other plans, environment operations, or internal actions. Internal actions are part of the plan body; they run internally in the agent and are not intended to change the environment. This is in contrast to actions, which change the world in which agents operate. The framework lets the agent programmer create internal actions. This means that the agent programmer can extend the capabilities of the agent by defining other customized functions. Customized internal actions can be used within the plan to perform actions.

As shown in Fig. 1, if no plan is considered for the current situation, the agent does not reach the goal, and if no *recovery plans* are given, the agent stops its work, otherwise it continues execution and reaches the goal.

Dynamic environments are difficult to control, programmers often cannot identify all the requirements and functions to manage the context. As far as the reasoning system can manage plans and adapt the agent to the context, it is not able to find a solution if there is no information about how to deal with unknown situations. This means that the plan selection process does not return solutions and the agent cannot achieve its goals.

More details about the reasoning system, the *Jason* framework, and how it works can be found in [1], [4].

III. NARS: Non Axiomatic Reasoning System.

NARS is an Artifial General Intelligence (AGI) [6] System developed in the context of an inferential system. It uses *Non-Axiomatic Logic*, a term logic that extends Aristotelian logic and its syllogistic forms to include compositions of terms as well as a notion of indeterminacy. It allows for various types of inference, including deduction, induction, and abduction [6], [10].

These types of inference allow the system to perform reasoning activities corresponding to planning (mostly deduction), learning (mostly induction), and explanation (mostly abduction), and to act autonomously. A prototype of the system, OpenNARS, has been implemented and tested in a variety of domains requiring, for example, Procedure Learning, diagnostics, Question Answering, and Anomaly Detection. [11]. In particular, the use cases tested include Procedure Learning tasks such as Pong, Real-Time Anomaly Detection (jaywalking and Pedestrian Danger) and Question-Answering in the Street Scene Dataset,as well as robot control in the TestChamber simulation developed for OpenNARS.

NARS is based on the belief that a key feature of intelligence is the ability to adapt under insufficient knowledge and resources.

The NARS is therefore designed to have the following characteristics:

- **Finite** - The information processing capacity of the system is limited, both in processing speed and in storage capacity: this means that not all options can always be considered or stored. The bottom line is that the system must weigh what to focus on and what to hold in memory.
- **Open** - problem-relevant information may not yet be available, and information can enter the system at any moment while the system is running and solving problems.
- **Realtime** - all tasks can be stopped at any moment by more urgent tasks. While the system is reasoning, the passing of time needs to be taken into account.

To make good use of available resources (both in terms of CPU speed and memory), the system needs a way to control its computational activities so that activities can run in parallel and at different speeds. This “Attentional Control” is implemented by a priority-based control strategy. This also avoids the bad consequences of combinatorial explosion in computation, as it forces the system to focus on task-relevant knowledge.

To implement this, the system uses a priority queue that allows higher priority items to be selected with higher probability than lower priority items. When the data structure is full and a new element is inserted, the element with the lowest priority is removed from the data structure. This data structure is also called *Bag* and is the key component of the system’s memory structure.

A term logic called “Non-Axiomatic logic” is used for argumentation. This logic is based on the idea that any statement in the system can gather both positive and negative evidence depending on how true the system believes it to be, and that these two values of evidence represent the truth of the statement.

Based on the previous description, the following relevant definitions are motivated, where instances, properties, propositions, and conjunctions are called “Compound Terms” (note

that there are more Compound Terms in NARS, but here are the ones relevant to our approach):

- **Term** - An identifier such as. *cat*.
- **Instances** - Terms marked as instances, such as. $\{key1\}$.
- **Properties** - Terms marked as properties, such as. *[green]*.
- **Statement** - A special case of a term that relates two terms to each other, via an inheritance relationship (e.g.: Cat is an animal, (cat $\rightarrow$ animal), or an implication (e.g., seeing light leads to hearing thunder: $((light \rightarrow [seen]) ==> ((thunder \rightarrow [heard])$)
- **Conjunctions** - Contains sequences of statements that, for our purposes, usually encode a temporal order, e.g. $((lighting \rightarrow [seen]), ((thunder \rightarrow [heard]))$,
- **sentence** - A statement associated with a truth value that is a tuple of positive and negative evidence (w_+, w_-).
- **Event** - A special case of a proposition to which an occurrence time is attached as an additional value.
- **Operation** - A special case of an event that the system itself can trigger whenever it wants. When triggered, an associated procedure is invoked, which for instance can control an actuator.
- **Task** - A record that has a priority value associated with it, and a value, Durability, that specifies how quickly the priority of the task should decrease. Tasks can be either Belief, Question, or Goal.

Summary of evidence is called "revision"; it allows, in the case of two sets of premises with the same notion, to the positive and negative evidence to obtain a conclusion that combines the evidence of both premises and leads to a "stronger" conclusion. Also, plans in Jason's sense are sentences in NARS, where usually for each plan step an operation op is performed under a certain circumstance a and an intermediate result c is expected. This is represented by the sentence $(a, op) \Rightarrow c$, which we call the "procedural hypothesis".

Whenever the circumstance a occurs and op has been executed, the system will predict c. When c is observed, positive evidence for the sentence is added, while when c does not occur as predicted, negative evidence is added. Clearly, the successful hypotheses are those that have much more positive evidence than negative evidence, and that have many pieces of evidence at the same time.

This is formalized by translating the tuple (w_+, w_-) into a tuple (f, c), where $f = \frac{w_+}{(w_+, w_-)}$ is Frequency and $c = \frac{(w_+ + w_-)}{(w_+ + w_-)+1}$ is Confidence. Frequency intuitively corresponds to the frequency with which this relationship exists within the samples collected, and Confidence describes the size of the sample space so far. Based on these terms, an overall expected truth value can be defined as $(c * (f - 0.5f) + 0.5f)$, which measures the overall degree of truth based on frequency and confidence.

Hypotheses with a high truth-expectancy value are precisely those hypotheses that are given a high priority value in the system by enforcing a positive correlation between truth and priority (the priority of a task is instantiated with the truth-expectancy of its sentence, although other factors may also play a role, such as the complexity of the link, but for our purposes they are secondary [5]). Therefore, they are not removed from the Bag, but are usually chosen to make a prediction, exactly as desired.

This serves a larger picture: Namely, when circumstances change, plans can become ineffective, and the solution must be able to deal with that. With the mechanism just described, NARS has the ability to notice this in order to adjust the truth value of the relevant knowledge downward (in the sense of adding negative evidence through revision). The mechanism that adds positive evidence when a prediction is successful is also the mechanism that builds the hypothesis in the first place. Essentially, there are three relevant inference rules in NAL that make this possible:

- **Temporal Intersection** - Formation of a sequence: From an event a, and an event b happening after a, form event (a, b). (note: b can as well be an operation, as in case of procedural hypotheses!)
- **Temporal Induction** - Formation of a hypothesis: From event a, and b, form hypothesis $a \Rightarrow b$, causing revision if the hypothesis is already in memory.
- **Deduction** - Allows to chain existing implication statements together. From $a \Rightarrow b$, and $b \Rightarrow c$, $a \Rightarrow c$ can be derived (note: this also works for the Inheritance relation). The merging of hypotheses based on preconditions is possible too, such as $(a, b) \Rightarrow c$, $(c, d) \Rightarrow e$, leading to $(a, b, d) \Rightarrow e$. Other forms of Deduction exist also (see [6]), but are secondary for our purposes here.
- **Prediction and Anticipation** - From observing event a, and hypothesis $a \Rightarrow b$, predict b, and add negative evidence to the hypothesis when b won't occur, using Revision. Technically, the derivation of the predicted event happens via the NAL deduction rule [6], and the negative evidence is created based on the mismatch between predicted and actual input.
- **Revision** - When $a \Rightarrow b$ is created from a new example, the evidence will be added to the hypothesis $a \Rightarrow b$ already in memory. Here, the positive and negative evidence will both be summed up separately. Hence, Revision is both responsible for strengthening and weakening a hypothesis, dependent on its success to predict correctly.

From a control perspective, the process is usually divided into two parts, both of which select two premises from memory to derive new information using inference rules.

- **Temporal Inference Control** : When a new event a enters the system, perform temporal intersection and temporal induction with existing events to derive new sequence events and new prediction hypotheses. This is done by sampling events from Event Bag. A Bag contains only input events and their sequences. That is, the derived sequences go into the event bag. The inferred hypotheses, on the other hand, go into the main memory, which is described below. Therefore, this inference can be described as follows:

  1) Process new incoming event, then repeat k (hyper-parameter) times:

a) Take an event from the event bag (prefer high priority elements, as explained before).
b) Apply inference rules between new event and taken outevent.
c) Put derived sequences in event bag, derived implications in main memory.

2) Place new incoming event in event bag as well as in main memory.

- **General Inference Control**: operates on the concept Bag, which is usually applied a fixed number of times for a fixed time interval. Concept Bag groups tasks into concepts according to their term, where each concept also stores tasks in a Bag. Thus, concepts compete for attention in Concept Bag, but tasks also compete for attention in each concept. Moreover, concept a has a link to concept b if they share a common term. Term-based linkage allows a newly derived task to effectively trigger revision when the related concept is already present in memory (gaining evidence for beliefs), which also increases the priority of the related concept. Moreover, concept-based linking allows concepts, when selected, to select not only a task per se, but also a belief about a concept to which they are semantically related, as a second premise. The selection process proceeds as follows:
  1) Select a concept from Concept Bag
  2) Select a task from the selected concept
  3) Select a belief from a neighbour concept
  4) Apply inference rules to task and belief
  5) Place derived and input tasks in the concept corresponding to their term (creating the concept if it does not yet exist and raising its priority), and trigger the revision for beliefs if there is already a belief in that concept. For goal processing, there is one more step
  6) Goal Processing / decision-making: For concept G, when goal g is processed, the concepts of the form $(a, op())are. \rightarrow G$, selecting the candidate that has the highest truth expectation and satisfies a according to Event Bag. Then, the operation $op()$ of the best candidate is executed. This is the decision making that OpenNARS uses, it considers both the truth of the procedural hypothesis and that the context is actually satisfied by checking with Event Bag.

While Temporal Inference Control can be seen as a way to control the perception and extraction of correlations in the input, General Inference Control allows knowledge to be combined in novel ways, e.g. by *deduction*.

These ideas, originally proposed in [11], have been implemented in OpenNARS and are sufficient to explain the role of NARS in our integration with the *Jason* planner. A key property of NARS that our integration exploits is that NARS can revise and withdraw failed plans when the success rate becomes too low, and new procedural knowledge can be added to the plan library as AgentSpeak plans, ultimately allowing AgentSpeak to be used in a less constrained way for autonomous systems facing changing environments. Note that a new plan cannot always be easily found, and in the worst case, when no information at all can be derived to suggest a relevant operation, "Motor Babbling", the random invocation of applicable actions, is necessary (or a user message must be generated if this is ineffective for too long). To implement listening to events, it is also necessary for NARS to constantly listen to the changing beliefs of the *Jason* system so that it can extract relevant correlations between sensorimotor events that can be stable enough to become part of useful procedural knowledge (effectively plans for *Jason*, after translation).

IV. Dynamic planning, *Jason* and NARS work together

A *Jason* agent consists of beliefs, goals, plans, internal actions and recovery plans. A *Jason* agent is programmed to reason using a library of plans and some information appropriate to the context. The main elements of NARS are beliefs and goals, which are used to create a plan.

As mentioned above, with the agent control cycle of *Jason* alone (Fig. 1), the agent programmer cannot make the agent follow the goal. At best, he must find a valid alternative to let the agent continue its execution. One way to overcome this limitation is to provide some "recovery plans", but even this cannot fix the problem and does not let the agent reach the goal.

We analyzed algorithms and planning systems [9] used in BDI planning theories to achieve our purpose, and the control cycle (shown in Fig. 1) was revised to allow the combination between Jason and the external reasoning system to support plan generation at runtime.

The Jason agent's intelligence is limited to the programmer's ability to predict all possible exceptions, so assistance from an external reasoning system allows the agent to rely on the latter's experience. Instead, NARS uses the knowledge gained from the environment and attempts to create a valid plan to achieve the agent's goal. The mechanisms behind the reasoning system reside in the inference engine, which uses non-axiomatic logic (NAL) [6], as we explained in the previous section.

NARS accepts plans, beliefs, and goals from the agent through appropriate functions that translate the AgentSpeak formalism into the Narsese language. This is possible because the first-order logic used by AgentSpeak and the non-axiomatic logic used by NARS are term logics.

The model we propose to implement a new control cycle that takes into account both Jason (BDI logic) and NARS is shown in Fig. 2. It is a customized control cycle that maintains the efficiency of the original control cycle, but supports the NARS inference system as an external service that helps the *Jason* planner when needed. The *Jason* agent finds an applicable plan in the plan library. This phase is identical to the standard control cycle of the *Jason* agent. The process of finding an applicable plan is through a unification process and a context check on the plan's precondition. If an applicable plan is found, then the *Jason* agent continues its process until the goal is reached. If no plan is found, the *Jason* agent starts

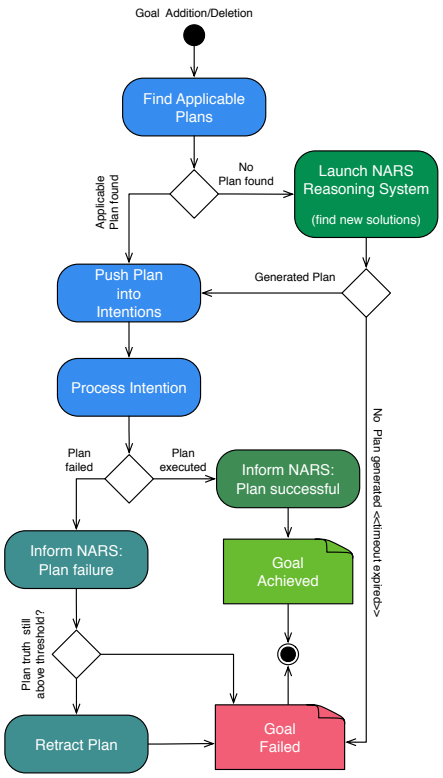

Fig. 2. The standard AgentSpeak control cycle merged with OpenNARS underpinning process.

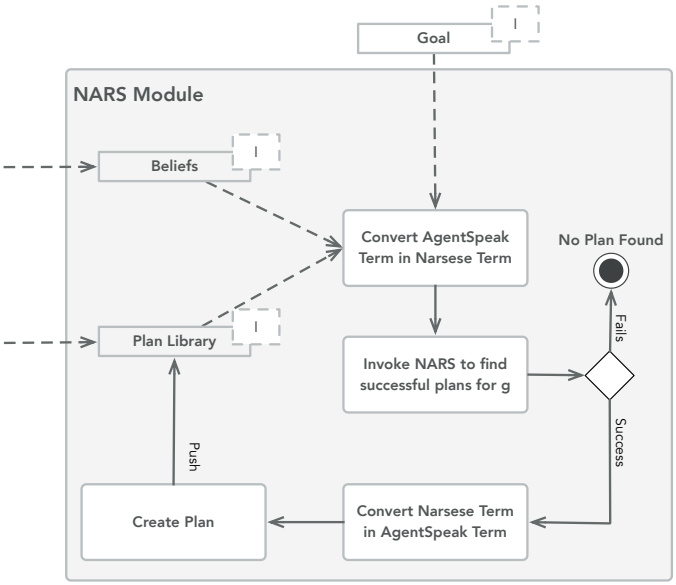

Fig. 3. A view of components for the NARS Module.

NARS through an internal action called *invokeNARS* to find a solution. At the end of the NARS reasoning process, a new plan is generated and sent to the agent *Jason*. The agent *Jason* continues its reasoning cycle and pushes the plan into the intent queue. Once the new intention is ready for execution, the agent processes it. If NARS does not find an alternative plan, the goal has failed.

Depending on the result of the plan execution, the agent informs NARS about the success or failure of the process. The messages sent by the agent let NARS revise the truth value computed by NAL to estimate the goodness of the plan. A successful plan increases the truth value and vice versa. If the truth value of the plan falls below a fixed threshold, NARS informs the agent *Jason* to withdraw the plan. Defining a threshold value[1] allows avoiding the generation of the same plan by NARS. In fact, the NARS inference engine and NAL work in such a way that they always generate the same plan starting from the same resources. Obviously, this is not useful for the problem we have. We need to force the inference engine to store information about unsuccessful plans to avoid their re-creation.

In the next section, we detail the NARS module that implements *launch NARS Reasoning System* in Fig. 2.

A. The NARS Module

The main goal of the NARS module is to provide alternative plans, if any, to achieve an agent's goal. The module (see Fig. 3) begins by converting beliefs, plans, and goals from AgentSpeak to Narsese. Once the conversion is complete, the module invokes the NARS reasoner to find a plan using NAL-based inference. The latter uses non-axiomatic logic operations to perform the finding process. If no solutions are generated, the NARS planner fails and no plans are added to the plan library. If it succeeds, the result of the NARS planner is converted from Narsese to AgentSpeak. Once the conversions are complete, the agent creates a new plan that is added to the plan library. Once the agent pushes the plan into the plan library, the control cycle continues with the agent's reasoning cycle using the new plan. The agent transforms the added plan with the intent that it will be processed and turned into an action to achieve the goal (as shown in Fig. 2) .

1) Implementation details.

From an implementation point of view, we have added some modules to the Jason framework. Fig. 4 shows a structural view of the extended version of Jason. The dashed square represents the Jason framework and the green blocks represent the modules we added. We have only detailed the part of the framework that is useful to show how we implemented the combination with NARS. In particular, the class *invokeNARS* implements an internal action using the interface provided by the *Jason* framework. The class *NARSEngine* implements all the communication mechanisms between the *Jason* agent and NARS that we described in this section.

The use of an external reasoning system is guaranteed by the correct internal action definition. The agent programmer's task is to write the agent's code using all the resources given by Jason. If he does not know how to program the agent's behavior, i.e., if at design time he has no clue how to write a plan to achieve a goal, he can resort to external reasoning without changing the logic of his application. The functionality of an external reasoning system is invoked via the internal action. Thus, if the programmer is not able to predict all

[1]The threshold is set to a static value defined at design time. We are working on making this value dynamic by gaining information through the agent's experience

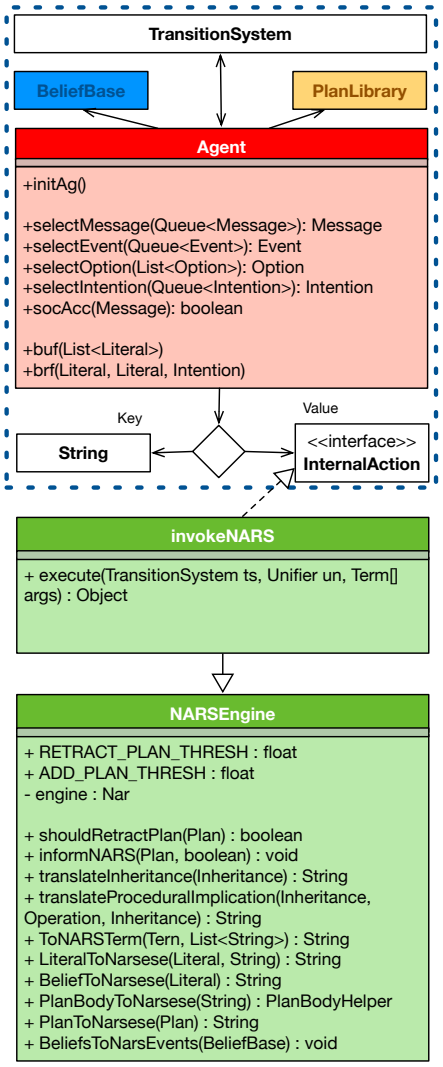

Fig. 4. A class diagram reporting the structure of the enriched *Jason* framework

possible scenarios in the unknown environment, he implements the internal action and only during the execution phase the reasoning system is invoked. This fact represents a great strength and a valuable contribution. The system programmer does not have to worry about the uncertain and unknown situations, all the agents he programs still have the same structure even if the reasoning system is invoked. The complexity of the code he writes does not increase when the external reasoning system is invoked, and most importantly, the programmer does not have to intervene. *NARSEngine* works in the background. We illustrate this in the following section.

V. Example of coding *Jason*+NARS

We made several proofs to validate our approach. We used JaCaMo[2] v0.0.7b [12] and OpenNARS[3] 3.0.2 [5]. OpenNARS is the open source version of NARS and JaCaMo is a framework for programming multiagent systems that combines *Jason* with two other important technologies, CArtAgO [13] and Moise [14].

We experimented with several scenarios in which agents had to reorganize their plan library to accomplish their assigned task. Below we briefly illustrate the code of a simple example. Our goal is only to show the functionalities provided by the control cycle shown in Fig. 2. Our goal is to illustrate how and when a plan is added or withdrawn.

In our example, we consider failed plans and have the system remove them when the truth value of the plan used falls below a threshold. For the truth value, we used the one defined internally by NARS:

$$\tau = (c * (f - \frac{1}{2}) + \frac{1}{2})$$

where c is confidence and f frequency, see [6].

In the scenario presented in this section, a robot works in a kitchen and has to serve a user by warning or advising him. For example, if the pantry is about to empty, the robot is supposed to report the situation and suggest what to buy. In this section, we report only a small illustrative part of the code generated for deploying agents in the robot.

The following part of the code represents the standard code for an agent.

```
//Beliefs
position(kitchen).
//Desires:
!serve_user.
//Plans:
@critical
+!serve_user : position(kitchen)
             <- .print("I_am_in_the_kitchen").
-!serve_user <- set of recovery plans.
```

Here we consider the subgoal related to the positioning of the robot in the kitchen. It is the first subgoal to start everything else in the kitchen. Belief refers to the position of the robot in the kitchen. The desire is "serve the user", and to achieve it, the plan has the context *position (kitchen)*, which must be true. If the context is true, the associated action, e.g. *.print("I am in the kitchen")*, is executed and the goal is achieved. In case the context is false, the programmer should have written one or more recovery plans. If no recovery plan is applicable, the target fails and the agent aborts its execution.

In our approach, the list of recovery plans may instead contain the internal action *invokeNARS*.

```
//Beliefs
position(kitchen).
//Desires:
!serve_user.
//Plans:
@critical
+!serve_user : position(kitchen)
             <- .print("I_am_in_the_kitchen");
                jia.invokeNARS(critical, true).
-!serve_user : contexts <- set of recovery plans.
-!serve_user <- jia.invokeNARS(critical, false).
```

In the above code we have added only two lines: *jia.invokeNARS(critical, true)* and *jia.invokeNARS(critical, false)*

Plans in Jason are exectuted in sequence and this guarantees that NARS is invoked if, and only if, there are no alternatives. *jia.invokeNARS(critical, ...)* is an internal action, it invokes the NARS engine with two options:

[2]http://jacamo.sourceforge.net/
[3]https://github.com/opennars/opennars

- *jia.invokeNARS(critical, true)*, the Jason plan is applicable and NARS does not need to find an alternative plan. NARS increments the truth value of the plan because the goal is achieved.
- *jia.invokeNARS(critical, false)*, is activated when all designed plans fail (both plans and recovery plans). The NARS engine starts generating plans. Using the same line of code, NARS attempts to generate a plan and lowers the truth value of the failed plan (serve_user). If the plan's truth value falls below the specified threshold after a certain number of failures, NARS withdraws the plan from the plan library.

Therefore, the first part of the code implements the plan at design time; the programmer of the agent has nothing to program more. The second part implements the agent's ability to execute at runtime. As said, this is the main strength of the proposed approach: the programmer does not have to write any additional code, the structure of the agent, its behaviour and mechanisms, have not been changed. Deliberation and runtime scheduling are hidden in the internal actions of *invokeNARS*, whose code is not shown here for this example, but the implementation of the classes *invokeNars* and *NARSEngine* discussed in the paper.

invokeNARS effectively implements the combination between the *Jason* Framework and OpenNARS.

VI. Related Work

Attempting to give agents a high degree of autonomy is an endeavor that many researchers are pursuing. Finding a way to plan at runtime allows agents to exhibit autonomy and adapt to changing contexts. One of the most efficient, but also one of the most rigid, ways to program agents is to specify the actions they must perform based on a known set of plans. At most, an agent can be enabled to assemble plans based on their pre- and post-conditions. Agents have no insight into the plans they execute; they use them as a black box and know only their preconditions.

Jason, like other frameworks and agent interpreters, is based on the concept of choosing a plan. Reasoning about the state of the environment leads to plan selection. The selected plan may fail, and the agent infers that the goal was not achieved. Other Jason-like implementations, such as Jack [15], execute alternative plans in the plan library until the goal is reached. If there are no alternative plans, the goal is not reached.

In all these cases, the agent is unable to independently and completely repair situations for which no plans have been provided to pursue a goal.

Meneguzzi and Luck in [16] propose a procedural agent model that can be modified to create new high-level plans starting from the fine-grained plans of the plan library. They modify the agent model. In our approach, this does not happen, which has the advantage of not increasing the complexity in terms of programming and execution. The agent behaves reactively, the reasoning cycle is modified as needed, and the agent's execution is not interrupted. Moreover, Meneguzzi and Luck solve the problem by composing the plans. They always start from the plan library and can only compose from the existing plans that have the same context. We use NARS and connect Jason to it as Meneguzzi and Luck do with STRIPS, but with the difference and advantage that with NARS the overall system finds a new recovery plan with fewer resources and information. While STRIPS generates and proves all possible plans that can be composed, NARS generates plans with non-axiomatic logic. Thus, it generates fewer plans from less information.

A similar approach is taken by De Silva et al. with the same limitation. Moreover, STRIPS is constrained by knowledge about the environment: resources, context, and actions must be known. In our approach, NARS requires less information about the environment; in fact, it works under the assumption of low resources and knowledge. Therefore, our approach is more flexible and able to find more solutions.

Hubner et al. in [17] use declarative goals to represent the state of the world when the goal is achieved. In this way, they make the definition of the goal implicit within the programming language (AgentSpeak). Using patterns, authors can then translate goals directly into code. In this way, they leave the agent structure unchanged, but do not go beyond three types of declarative goals. They also crystallize the possibility of creating new plans to a minimal number. Hubner et al. provide a simple tool to create new plans, but they maintain the limit of stopping the agent if it does not find a useful plan.

NARS natively performs an evaluation of plans. The ability to add and withdraw plans is beneficial and increases the strength of our approach. In this way, the plan library is continuously updated based on changes in the environment, allowing the agent to better deliberate during execution.

However, a limitation may be that when using NARS, we remain dependent on the ability to store new plans, the size of the memory, bag and the time required to test the plans.

VII. Discussion and Conclusion

Nowadays, multi-agent programming is a useful paradigm for solving problems related to complex systems and runtime planning. In this paper, we focus on runtime planning, i.e., the creation or composition of new plans when agents realize preprogrammed plans that have proven unsuccessful.

We use the BDI agent paradigm and the *Jason* framework to develop multiagent systems capable of dealing with situations for which no plans have been provided to pursue a goal. Although the use of deliberative agents can be useful for solving the above problems, *Jason* has not proven to be flexible enough for runtime scheduling alone, especially when the plan library requires adaptation.

In this work, we developed a model for a BDI reasoning cycle using an extended reasoning system. We chose to use NARS [5] to support the planning activities. The efficiency of NARS for procedural knowledge learning was demonstrated using the mechanisms described in [11]. NARS provides a robust reasoning cycle in situations where robots need to perform autonomous actions and where it is crucial to find alternative solutions to those defined at planning time.

Even though *Jason* is able to operate in defined situations, the creation of the new control cycle has improved the ability

of each *Jason* agent to find a solution to achieve a goal. In addition, the new control cycle, shown in Fig. 2, has not changed the *Jason* reasoning cycle. This is a big advantage for developers writing programs with *Jason*. They only need to apply the new internal action where it is needed, without breaking the code flow.

The novelty introduced is the ability of the system to be autonomous, in the sense that the system can find a plan (which is not known a priori) based on its, possibly acquired, knowledge. This can be applied in different types of complex domains: Human-Robot Interaction, IoT and Intelligent* Systems, Swarm, to name a few. In all these contexts, the ability to change from within is one of the key challenges.

Overall, the proposed method can greatly increase the autonomy of systems and make them act better even in scenarios where unexpected circumstances cause existing plans to fail, or where new plans are needed to deal with novel situations.

The tests we have conducted show that our approach is a viable way to allow for goal-directed planning with adaptation of behaviors based on plan success. However, there are still some shortcomings in using NARS, which we will face soon. NARS operates with limited memory, so we cannot be sure that all generated plans are profitably stored and sent to the plan library.

In the near future, we plan to conduct experiments that will allow us to quantify the response times of our reasoning cycle and the amount of plans generated under the same conditions compared to standard approaches.

REFERENCES

[1] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. USA: John Wiley & Sons, Inc., 2007.

[2] A. S. Rao, M. P. Georgeff *et al.*, "BDI agents: from theory to practice." in *ICMAS*, vol. 95, 1995, pp. 312–319.

[3] M. Wooldridge, *Reasoning about rational agents*. MIT press, 2003.

[4] R. H. Bordini and J. F. Hübner, "BDI agent programming in agentspeak using jason," in *International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 2005, pp. 143–164.

[5] P. Hammer, T. Lofthouse, and P. Wang, "The opennars implementation of the non-axiomatic reasoning system," in *Artificial General Intelligence*. Springer, 2016, pp. 160–170.

[6] P. Wang, *Non-axiomatic logic: A model of intelligent reasoning*. World Scientific, 2013.

[7] M. Bratman, *Intention, plans, and practical reason*. Harvard University Press Cambridge, MA, 1987, vol. 10.

[8] A. S. Rao, "AgentSpeak (l): BDI agents speak out in a logical computable language," in *European workshop on modelling autonomous agents in a multi-agent world*. Springer, 1996, pp. 42–55.

[9] F. Meneguzzi and L. de Silva, "Planning in bdi agents: a survey of the integration of planning algorithms and agent reasoning," *Knowledge Eng. Review*, vol. 30, pp. 1–44, 2015.

[10] P. Wang, *Rigid flexibility: The logic of intelligence*. Springer Science & Business Media, 2006, vol. 34.

[11] P. Hammer and T. Lofthouse, "Goal-directed procedure learning," in *International Conference on Artificial General Intelligence*. Springer, 2018, pp. 77–86.

[12] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013.

[13] A. Ricci, M. Viroli, and A. Omicini, "CArtAgO: A framework for prototyping artifact-based environments in mas," in *International Workshop on Environments for Multi-Agent Systems*. Springer, 2006, pp. 67–86.

[14] M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat, "Moise: An organizational model for multi-agent systems," in *Advances in Artificial Intelligence*. Springer, 2000, pp. 156–165.

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

[16] F. Meneguzzi and M. Luck, "Composing high-level plans for declarative agent programming," in *International Workshop on Declarative Agent Languages and Technologies*. Springer, 2007, pp. 69–85.

[17] J. F. Hübner, R. H. Bordini, and M. Wooldridge, "Plan patterns for declarative goals in agentSpeak," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, 2006, pp. 1291–1293.

Francesco Lanza Francesco Lanza has a PhD in Technological Innovation Engineering from the University of Palermo. His research is on multi-agent systems, cognitive robotics, natural language processing, computer vision and knowledge engineering. His goal is to develop cognitive systems for robotic platforms capable of interacting with humans.

Valeria Seidita Valeria Seidita is a Assistant Professor at the University of Palermo. She received her master's degree in Electronic Engineering in 2004 from the University of Palermo. In 2008 she obtained her PhD in Computer Science from the same university with a thesis on "A Situational Method Engineering based Approach for Constructing Multi-Agent System Design Processes: Techniques and Tools". Since December 2011 she is Assistant Professor. Valeria Seidita does research in the field of software engineering. She is particularly interested in theories, techniques, and practises for engineering complex and intelligent systems with emergent functionalities. She has published over 60 papers in international journals and conferences and is a reviewer for several international journals as well as a member of the program committee of numerous international conference workshops.

Patrick Hammer Patrick Hammer is an AGI researcher and currently research assistant at Temple University, Philadelphia. His main research interests include Non-Axiomatic Reasoning System (NARS), specific implementation designs thereof, and to improve their real-time reasoning learning capabilities to enhance the autonomy of intelligent agents. Among his goals is to replicate, in computer systems, key aspects of natural intelligence as showcased by higher-developed animals in their cognitive capacities and ability to improvise. Previous works include research in temporal, procedural and introspective reasoning, and the "OpenNARS for Applications" design and implementation.

Pei Wang Pei Wang is an Associate Professor at Temple University. He received his Ph.D. in Computer Science and Cognitive Science from Indiana University at Bloomington, and his M.S. and B.S. in Computer Science, both from Peking University. His research interests include artificial intelligence and cognitive science, especially on general theory of intelligence, formal models of rationality, reasoning under uncertainty, learning and adaptation, knowledge representation, and real-time decision making. He is the Chief Executive Editor of the Journal of Artificial General Intelligence.

Antonio Chella Antonio Chella is a Professor of Robotics at the University of Palermo, head of the Robotics Laboratory and director of the Computer Engineering Section of the Department of Engineering. He is a former director of the Department of Computer Engineering and of the Interdepartmental Center for Knowledge Engineering. He is an Honorary Professor in Machine Learning and Optimization at the School of Computer Science, University of Manchester, UK. He is a member of the Italian National Academy of Sciences, Letters and Arts in Palermo, Italy. He received the James S. Albus Medal of the BICA Scientific Society. He coordinated research projects from Italy, Europe, and the United States. He is author or co-author of more than 200 publications in the fields of robotics and artificial intelligence. His current research topic concerns the interdisciplinary field of AI and machine consciousness, involving multiple disciplines, including computer science, psychology, cognitive science and philosophy. Prof. Chella's scientific research activities have been the subject of articles and interviews that appeared in national and international journals and newspapers.