A BDI Learning Agent for Environments with Changing Dynamics

Paper 306

ABSTRACT

We propose enhancements to a framework that integrates learning capabilities to improve plan selection in the successful and popular Belief-Desire-Intentions agent programming paradigm. In learning which plan to select, a crucial issue in the online setting is how much to trust what has been learnt so far (and therefore exploit it) versus how much to explore to further improve the learning. In this paper we construct a confidence measure based on a previously used notion of stability in the outcomes observed for a particular plan, combined with a consideration of the extent to which new worlds are being witnessed by the plan. This new measure dynamically adjusts based on agent performance, allowing in principle, infinitely many learning phases. Additionally, it scales up irrespective of the complexity of the goal-plan hierarchy implicit in the agent's plan library. We demonstrate the utility of our approach with results obtained in a practical energy storage domain.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Intelligent agents; I.2.6 [**Learning**]: Induction

General Terms

Algorithms, Performance, Experimentation

Keywords

BDI agent-oriented programming, Learning (single and multi-agent)

1. INTRODUCTION

Agent systems built in the Belief-Desire-Intention (BDI) agent-oriented programming paradigm [8, 14, 22] do not traditionally do learning. However, if a deployed agent (i.e., agent system) is to be able to adapt over time to a changing situation, then learning becomes important. Our vision is to be able to deploy an agent system that is capable of adjusting to ongoing changes in the environment's dynamics in a robust and effective manner. Nonetheless, we still want to adhere to the BDI-style programming principle of leveraging on available procedural "know-how" information that is available. In that sense, we are *not* interested in agents that learn from scratch, but rather on agents that learn how to (better) use the existing operational knowledge of the domain they are situated in.

Cite as: A BDI Learning Agent for Environments with Changing Dynamics, Authors, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Yolum, Tumer, Stone and Sonenberg (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. XXX–XXX.

To that end, in this paper, by building on the recent work of [1, 18, 19], we develop mechanisms for which BDI-type agents can *adaptively learn* the appropriate selection of plans in a situation, as they go. One of the key issues in any learning system is that of exploitation vs. exploration, that is, how much to believe and exploit the current (learnt) knowledge, versus how much to try things in order to gain new knowledge. Important in this balance is an understanding of how much learning has already been done, or how much has already been explored. In [18, 19], a "coverage-based" measure of confidence was used to capture how much the agent should trust its current understanding (of good plan selection), and therefore exploit rather than explore. Intuitively, such confidence was based on the degree to which the space of possible execution options for the plan has been explored (i.e., covered) so far. The greater the extent to which this space has been explored, the greater the confidence, and consequently the more likely the agent is to exploit.

As recognised by [19], the coverage-based confidence approach does not support learning in a changing environment. This is because the confidence increases *monotonically* and, as a result, there is no ability for the agent to become less confident in its choices, even if its currently learned behaviour becomes less successful due to changes in the environment.

Consider, for instance, a smart office building equipped with a large battery system that can be charged when there is excess power, and used (i.e., discharged) when there is excess demand, in order to achieve an overall desired building consumption rate for a given period. A battery installation is built from independent modules with different chemistries. Initially, the battery controller can be programmed (or can learn) to operate optimally. However, over time, the modules in a battery tend to operate less well or some may even cease to function altogether. In addition, modules may be replaced with some frequency. Thus, what is needed is a controller agent that after having explored the space well and developed a high confidence in what it has learned, is able to observe when this learned knowledge becomes unstable, and dynamically modifies its confidence allowing for new exploration and revised learning.

In this work, we develop a new confidence measure, compatible with the BDI learning framework, which allows the agent to adjust its confidence as the environment changes. The new metric is built from two ingredients. First, it uses the notion of plan stability from [1, 19] to quickly estimate how much the different options for achieving a goal have been explored. Second, it considers how much the agent is experiencing states that have been seen before vs how much it is seeing new situations, by using a sliding window that checks what percentage of situations in the window have been seen previously. If a substantial number of new situations are being experienced, then the agent should be less confident in what it has

previously learnt.

The rest of the paper is organised as follows. In the following section, we provide an overview of the basic BDI learning framework on which this work is based, as developed by [1, 18, 19]. We then explain in detail our new proposal for a dynamic measure of confidence that may be used by the BDI agent at plan-selection time to continually adjust to a changing world. Following that, we describe an energy storage domain taken from a real application where the environment dynamics changes over time requiring adaptive learning. We then specify some experiments evaluating the battery controller agent in different situations, and demonstrate that our learning approach for BDI systems does in fact allow the agent to adjust in a variety of ways to an environment where battery behaviour changes. We conclude the paper by discussing related work and some limitations requiring future work.

2. THE BDI LEARNING FRAMEWORK

We begin by reviewing the basic agent programming framework that will be used throughout the paper [1, 18, 19], which is a seamless integration of standard Belief-Desire-Intention (BDI) agent-oriented programming [14, 22] with decision tree learning [11].

Generally speaking, BDI agent programming languages are built around an explicit representation of propositional attitudes (e.g., beliefs, desires, intentions, etc.). A BDI architecture addresses how these components are represented, updated, and processed to determine the agent's actions. There are a plethora of agent programming languages and development platforms in the BDI tradition, including JACK [6], JADEX [13], and Jason [3] among others. Specifically, a BDI intelligent agent systematically chooses and executes *plans* (i.e., operational procedures) to achieve or realize its goals, called *events*. Such plans are extracted from the agent's *plan library*, which encodes the "know-how" information of the domain the agent operates in. For instance, the plan library of an unmanned air vehicle (UAV) agent controller may include several plans to address the event-goal of landing the aircraft. Each plan is associated with a *context condition* stating under which belief conditions the plan is a sensible strategy for addressing the goal in question. Whereas some plans for landing the aircraft may only be suitable under normal weather conditions, other plans may only be used under emergency operations. Besides the actual execution of domain actions (e.g., lifting the flaps), a plan may require the resolution of (intermediate) sub-goal events (e.g., obtaining landing permission from the air control tower). As a result, the execution of a BDI system can be seen as a *context sensitive subgoal expansion*, allowing agents to "act as they go" by making *plan choices* at each level of abstraction with respect to the current situation. The use of plans' context (pre)conditions to make choices as late as possible, together with the built-in goal-failure mechanisms, ensures that the system is responsive to changes in the environment.

It is not hard to see that, by grouping together plans responding to the same event type, the agent's plan library can be seen as a set of goal-plan tree templates (e.g., Figure 1) [16]: a goal-event node (e.g., goal G_1) has children representing the alternative *relevant* plans for achieving it (e.g., P_a, P_b and P_c); and a plan node (e.g., P_f), in turn, has children nodes representing the subgoals (including primitive actions) of the plan (e.g., G_4 and G_5). These structures, can be seen as AND/OR trees: for a plan to succeed all the subgoals and actions of the plan must be successful (AND); for a subgoal to succeed one of the plans to achieve it must succeed (OR). Leaf plans are meant to directly interact with the environment and so, in a given world state, they can either succeed or fail when executed; this is marked accordingly in the figure for some particular world (of course such plans may behave differently in other states).

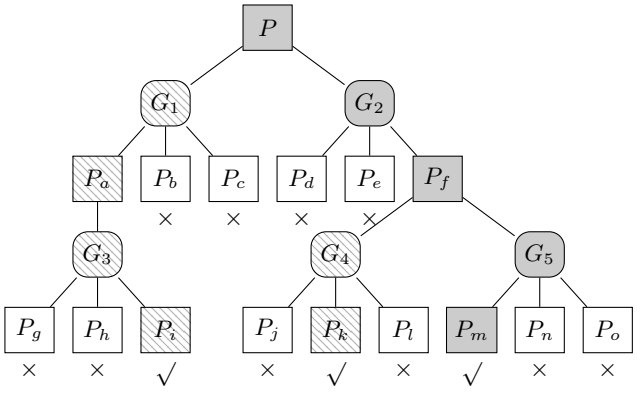


Figure 1: An example goal-plan hierarchy.

We will use the notion of an *active execution trace* previously introduced in [19] to capture each decision sequence that terminates in the execution of a leaf plan. Consider again the goal-plan structure of Figure 1 that shows the possible outcomes when plan P is selected in a given world w. Here, in order for the first subgoal G_1 to succeed, plan P_a must be selected followed by P_i that succeeds in the world (as indicated by the $\surd$ symbol). The active execution trace for this selection then is described as $\lambda_1 = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_i : w]$ (highlighted in Figure 1 as the line-shaded path terminating in P_i) where the notation $G[P : w]$ indicates that goal G was resolved by the selection of plan P given world state w. Subsequently subgoal G_2 is posted whose successful resolution is described by the intermediate trace $\lambda_2 = G[P : w] \cdot G_2[P_f : w'] \cdot G_4[P_k : w']$ followed by the final trace $\lambda_3 = G[P : w] \cdot G_2[P_f : w'] \cdot G_5[P_m : w'']$. Note that the world w' in λ_2 is the resulting world state from the successful execution of leaf plan P_i in the preceding trace λ_1. Similarly, w'' is the resulting world state from the execution of P_k in λ_2. There is only one way for plan P to succeed in the initial world w as described by the traces $\lambda_1 \ldots \lambda_3$. All other execution traces lead to failures (as depicted by the $\times$ symbol).

As can be seen, adequate plan selection is critically important in BDI systems. Whereas standard BDI platforms leverage domain expertise by means of *fixed* logical context conditions of plans, in this work, we are interested in exploring how a situated agent may *learn* or *improve* its plan selection mechanism based on experience, in order to better realize its goals. To that end, it was proposed to generalize the account for plans' context conditions to decision trees [11] that can be learnt over time [1, 18, 19]. The idea is simple: *the decision tree of an agent plan provides a judgement as to whether the plan is likely to succeed or fail for the given situation.* By suitably *learning* the structure of and adequately *using* such decision trees, the agent is able to improve its performance over time , lessening the burden on the domain modeller to encode "perfect" plan preconditions. Note that the classical boolean context conditions provided by the designer could (and generally will) still be used as initial necessary but possibly insufficient requirements for each plan that will be further *refined* over time in the course of trying plans in various world states.

Under the new BDI learning framework, two mechanisms become crucial. First, of course, a principled approach to learning such decision trees based on execution experiences is needed. Second, an adequate plan selection scheme compatible with the new

type of plans' preconditions is required. To select plans based on information in the decision trees, the work reported in [18, 19] used a probabilistic method that chooses a plan based on its believed likelihood of success in the given situation. This approach provides a balance between exploitation (by choosing plans with relatively higher success expectations more often), and exploration (by sometimes choosing plans with lower success expectation to get better confidence in their believed applicability by trying them in more situations). This balance is important because ongoing learning influences future plan selection, and subsequently whether a good solution is found.

When it comes to the learning process, the training set for a given plan's decision tree contains samples of the form $[w, o]$, where w is the world state—a vector of discrete attributes—in which the plan was executed and o is the execution outcome, namely, success or failure. Initially, the training set is empty and grows as the agent tries the plan in various world states and records each execution result. Since the decision tree inductive bias is a preference for smaller trees, one expects that the learnt decision tree consists of only those world attributes that are relevant to the corresponding plan's (real) context condition.

Due to the hierarchical nature of the goal-plan structure being executed by the agent (c.f. Figure 1), it is possible that the failure of a plan (e.g., P) is only due to a poor plan selection lower in the hierarchy (e.g., P_l is selected for goal G_4). To deal with this issue, the work in [1] uses a plan "stability" measure to take failures into account only when the agent is sufficiently sure that the failure was not due to poor sub-plan choices. To further understand this notion consider the case where plan selection results in the failed execution trace $\lambda = G[P : w] \cdot G_2[P_f : w_2] \cdot G_5[P_n : w_5]$.[1] What should we make of this failure from a learning perspective? Should the negative outcome be recorded for training our decision tree at non-leaf nodes P_f and P? The concern stems from the fact that these non-leaf plans failed not because they were a bad choice for world w but because a bad choice (P_n) was made further down in the hierarchy. To resolve this issue, the stability filter is used in [1] to record failures only for those plans whose outcomes are considered to be stable, or "well-informed."

Another approach reported previously is to adjust the plan selection probability based on some measure of "confidence" in the decision tree [18, 19] which considers the reliability of a plan's decision tree to be proportional to the number of sub-plan choices (or paths below the plan in the goal-plan hierarchy) that have been explored already: the more choices that have been explored, the greater the confidence in the resulting decision tree.

3. A DYNAMIC CONFIDENCE MEASURE

In this section we describe a dynamic confidence measure that may be used to guide exploration when learning plan selection using the framework described in Section 2. Conceptually, the value of the confidence measure relates to the degree of trust that the agent has in its current understanding of the world (from a learning perspective). Technically, recall that the confidence metric informs the agent about how much it should trust the outcome estimate provided by its current decision tree.

Our new confidence measure improves upon previously used measures in two important ways. Firstly, it caters to changing dynamics of the environment that often results in prior learning becoming less effective. The stability-based [1] and coverage-based [18, 19] measures that have been previously proposed do not support the requirement for adaptability to such changes. Moreover, the new measure proposed here subsumes the functionality of the former methods, as it behaves monotonically in environments where the dynamics are fixed. As such, it offers a direct replacement for the previous approaches. Secondly, the new measure does not rely on estimates of the number of choices in the goal-plan hierarchy as is the case in [18, 19], and scales to any general goal-plan hierarchy irrespective of its complexity.

To recap the definition of stability from [19]:

> *"A failed plan P is considered to be stable for a particular world state w if the rate of success of P in w is changing below a certain threshold."*

Our aim is to use this notion to judge how "stable" or well-informed the decisions the agent has made within a particular execution trace were. This is particularly meaningful for *failed* execution traces: low stability suggests that we were not well-informed and more exploration is needed before assuming that no solution is possible (for the trace's top goal in question). To capture this, we define the *degree of stability* of a (failed) execution trace λ, denoted $\zeta(\lambda)$ as the ratio of stable plans to total applicable plans in the active execution trace below the top-level goal event in λ. Formally, when $\lambda = G_1[P_1 : w_1] \cdots G_n[P_n : w_n]$ we define

$$\zeta(\lambda) = \frac{|\bigcup_{i=1}^{n} \{P \mid P \in \Delta_{app}(G_i, w_i),\ \mathit{stable}(P, w_i)\}|}{|\bigcup_{i=1}^{n} \Delta_{app}(G_i, w_i)|},$$

where $\Delta_{app}(G_i, w_i)$ denotes the set of all applicable plans (i.e., whose boolean context conditions hold true) in world state w_i for goal event G_i, and $\mathit{stable}(P, w_i)$ holds true if plan P is deemed stable at world state w_i, as defined in [19].

For instance, take the failed execution trace $\lambda = G[P : w] \cdot G_2[P_f : w_2] \cdot G_5[P_n : w_5]$ from before and suppose further that the applicable plans are $\Delta_{app}(G, w) = \{P\}$, $\Delta_{app}(G_2, w_2) = \{P_d, P_f\}$, and $\Delta_{app}(G_5, w_5) = \{P_m, P_n, P_o\}$. Further assume that P_d and P_n are the only plans deemed stable (in worlds w_2 and w_5 respectively). Then the degree of stability for the whole trace is $\zeta(\lambda) = 2/6$. Similarly, for the two subtraces $\lambda' = G_2[P_f : w_2] \cdot G_5[P_n : w_5]$ and $\lambda'' = G_5[P_n : w_5]$ of λ, we get $\zeta(\lambda') = 2/5$ and $\zeta(\lambda'') = 1/3$.

The idea is that every time the agent reaches a failed execution trace, the stability degree of each subtrace is stored in the plan that produced that subtrace. So, for our example, for plan P we store degree $\zeta(\lambda')$ whereas for plan P_f we record degree $\zeta(\lambda'')$. Leaf plan nodes, like P_n, make no choices so their degree is simply 1. Intuitively, by doing this, we record against each plan in the (failed) trace, an estimate of how informed the current (active) choices made for the plan were. Algorithm 1 describes how this (hierarchical) recording happens given an active execution trace λ. Observe how the stability measure is recorded against each plan in the trace: *RecordDegreeStability*(P, w, d) records degree d for plan P in world state w.

As a plan execution produces new failed experiences, the calculated degree of stability is appended against it each time. When a plan finally succeeds, we take an optimistic view and record 1 (i.e., full stability) against it. This, together with the fact that all plans do eventually become stable, means that $\zeta(\lambda)$ is guaranteed to converge to 1.

To aggregate the different stability recordings for a plan over time, we use the *average degree of stability* over the last $n \geq 1$ executions of plan P in w, denoted $\mathcal{C}_s(P, w, n)$. This provides us with a measure of confidence in the decision tree for plan P in state w. Intuitively, $\mathcal{C}_s(P, w, n)$ tells us how "informed" the decisions

[1] Note that trace λ assumes the successful resolution of subgoal G_1 and the resulting world state w', as described by λ_1 previously.

Algorithm 1: *RecordDegreeStabilityInTrace*(λ)

Data: $\lambda = G_1[P_1 : w_1] \cdot \ldots \cdot G_n[P_n : w_n]$, with $n \geq 1$.
Result: Records degree of stability for plans in λ.
if $(n > 1)$ **then**
  $\lambda' = G_2[P_2 : w_2] \cdot \ldots \cdot G_n[P_n : w_n]$;
  $d = \zeta(\lambda')$;
  RecordDegreeStability(P_1, w_1, d);
  RecordDegreeStabilityInTrace(λ');
else
  RecordDegreeStability$(P_1, w_1, 1)$;

taken when performing P in w were over the n most recent executions. Notice that if the environment dynamics are constant, this measure monotonically increases from 0, as plans below P start to become stable (or succeed); it reaches 1 when all tried plans below P in the last n executions are considered stable. This is what one might expect in the typical learning setting. However, if the environment dynamics were to change and plans start to fail or become unstable, then the measure behaves non-monotonically and adjusts confidence accordingly.

The stability-based confidence measure $\mathcal{C}_s(\cdot, \cdot, \cdot)$ would make a useful heuristic for exploration (i.e., plan selection) in its own right: when the confidence is at its lowest the agent does maximum exploration, and when it is at its highest, the agent fully utilises the decision trees. A problem with this approach, though, is that such measure only covers the space of *known* worlds. This means that whenever a new world is witnessed, this stability-based confidence will be zero, meaning that the agent will choose randomly. This is hardly beneficial since what we would really like is to use the learnt generalisations to classify (i.e. predict the outcome in) this new world rather than being agnostic about it. What is missing is a complementary metric that contributes to our net confidence but that is independent of w.

One way to address this limitation is by monitoring the rate at which new worlds are being witnessed by a plan P. During early exploration, it is expected that the majority of worlds that a plan is selected for will be unique, thus yielding a high rate and a low confidence. Over time, as exploration continues, the plan would get selected in all worlds in which it is reachable and the rate of new worlds would approach zero, while our confidence over this period increases to its maximum. So, we define our second confidence metric $\mathcal{C}_d(P, n) = |\mathit{NewStates}(P, n)|/n$, where $\mathit{NewStates}(P, n)$ is the set of world states that have been seen for the first time in the last n executions of P. Clearly, $\mathcal{C}_d$ is guaranteed to converge to 1 as long as all worlds where the plan might apply are eventually witnessed.

In summary, we have defined two confidence metrics over two orthogonal dimensions. Stability-based confidence $\mathcal{C}_s(P, w, n)$ is meant to capture how well-informed the last n executions of plan P in world w were, whereas world-based confidence $\mathcal{C}_d(P, n)$ is meant to capture how well-known the worlds in the last n executions of plan P were, compared with what we had experienced before.

We now have all the technical machinery to define our final (aggregated) confidence measure $\mathcal{C}$ that takes into account both the above metrics. Specifically, the overall confidence in the decision tree of plan P in world w relative to the last n experiences is defined as follows:

$$\mathcal{C}(P, w, n) = \alpha \mathcal{C}_s(P, w, n) + (1 - \alpha)\mathcal{C}_d(P, n),$$

where α is a weighting factor used to set a preference bias between the two component metrics.

Finally, we show how this confidence measure is to be used within the plan selection mechanism of a BDI agent. Remember that for a given goal-event that needs to be resolved, a BDI agent may have several applicable plans from which one ought to be selected for execution. The BDI learning framework described in Section 2 will chose probabilistically among these options in a way proportional to some given weight per plan—the more weight a plan is assigned, the higher the chances of it being chosen. Following [18, 19], we define this selection weight for plan P in world w relative its last n executions as follows:

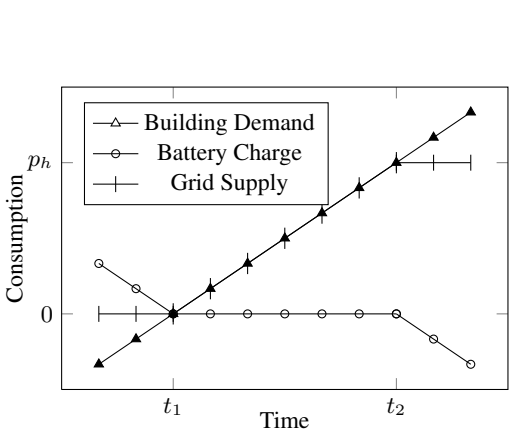
$$\Omega(P, w, n) = 0.5 + [\mathcal{C}(P, w, n) \times (\mathcal{P}(P, w) - 0.5)],$$

where $\mathcal{P}(P, w)$ is the probability of success of plan P in world w as given by P's decision tree. Indeed, this weight definition is basically that of [18, 19], except for the use of our new confidence metric $\mathcal{C}(\cdot, \cdot, \cdot)$ as defined above. The idea is to combine the likelihood of success of plan P in world w (i.e. the term $\mathcal{P}(P, w)$) with a confidence bias (in this case $\mathcal{C}(\cdot, \cdot, \cdot) \in [0.0 : 1.0]$) to determine a final plan selection weight $\Omega(\cdot, \cdot, \cdot)$. When the confidence is maximum i.e. 1.0, then $\Omega(\cdot, \cdot, \cdot) = \mathcal{P}(\cdot, \cdot)$, and the final weight equals the likelihood reported by the decision tree; when the confidence is zero, then $\Omega(\cdot, \cdot, \cdot) = 0.5$, and the decision tree has no bearing on the final weight (a default weight of 0.5 is used instead).

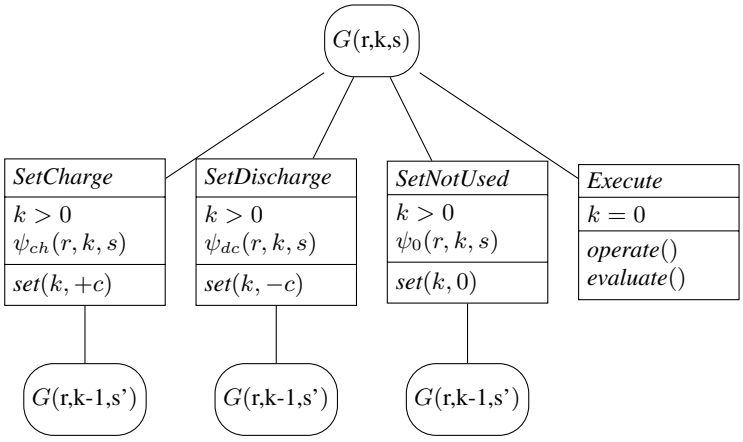
This mechanism for probabilistic plan selection using dynamic confidence, together with the decision tree integration explained in Section 2 completely describes our BDI learning framework. Given this, in the subsequent section we cover the use of this framework in the design of a complete BDI system for an energy storage scenario. The application is a fully functional implementation of a modular battery controller from initial specification. More importantly, by incorporating the learning framework, we shall demonstrate that the program is able to handle (certain) foreseeable changes in operational dynamics for the battery system once deployed.

4. AN ENERGY STORAGE APPLICATION

Energy storage enables increasing levels of renewable energy in our electricity system, and the rapidly maturing supply chains for several battery technologies encourages electricity utilities, generators, and customers to consider using large battery systems. Consider a controller for managing the overall energy demand of a smart office building comprising of a set of loads (e.g., appliances in the building), some renewable sources (e.g., solar panels on the roof and a local wind turbine), and a *modular battery system*. The building is connected to the main electricity grid, and economics govern that the overall grid power demand of the building be maintained within the range $[0 : p_h]$; see Figure 2(a). Since there is little control over the demand in the building, and certainly no control over the renewable generation, it is possible that the building power consumption falls outside this range for some period in the day. For instance, if the renewable generation is high relative to the building loads, then net consumption (Building Demand) may fall below 0 (e.g., period prior to t_1); whereas if demand is higher than generation, then the net building consumption may rise above p_h (e.g., period after t_2). While there is little that can be done about the net building consumption and generation, we do have control over the use of the battery system. Hence, by suitably ordering the battery system (Battery Charge) to charge (i.e., act as a load) or discharge (i.e., act as a generator) at determined rates, it is possible to influence the net demand and effectively the energy drawn from the electricity grid (Grid Supply).



(a) Use case scenario for a modular battery system.



(b) Goal-plan hierarchy for a k-modules battery system.

Figure 2: An energy storage application.

Large battery systems usually comprise of multiple modules that can be controlled independently [12]. Modules may be operated in synchrony, but there are often strategic reasons for keeping some modules in different states. For example, if it is undesirable to change the direction of power flow between charging and discharging too frequently, a subset of modules may be used for each direction until it is necessary to swap their roles. Also, some technologies have specific requirements, such as zinc-bromine flow batteries needing complete discharges at regular intervals to "strip" the zinc plating and hence ensuring irregularities never accumulate. Where they exist, such requirements place further constraints on module control.

So, given a large battery installation, we are interested in a mechanism to achieve a desired rate of charging or discharging, by suitably setting each module in the battery—the overall rate being the sum over the modules' rates. For simplicity, we assume homogeneous capacity c of the modules (but with possibly different chemical properties and constraints), and hence an overall system capacity of $c \times k$ (where k is the total number of modules in the system). Each module, in turn, may be configured as charging ($+c$), discharging ($-c$), or not in use (0). By appropriately setting each module's operational state, the total response of the battery system may be adjusted in steps of $\pm c$. While hardwired control is indeed possible, it is not ideal due to the fact that battery performance is susceptible to changes over time—modules tend to lose actual capacity—and may diverge from normal. What is required is an *adaptable* control mechanism that accounts for such drift.

Figure 2(b) depicts a BDI controller for this application. Top-level goal-event $G(r, k, s)$ requests a battery charge/discharge (normalized) rate of $r \in [-1.0 : +1.0]$, where -1.0 (1.0) indicates maximum discharge (charge) rate, s stands for the current state of the battery system as per sensor readings, and k is (initially) the number of modules in the system. The BDI controller works by recursively configuring each module for the period in question using the plans *SetCharge* (charging at rate $+c$), *SetDischarge* (discharging at rate $-c$), and *SetNotUsed* (disconnected), and finally, after all modules have been configured, physically operating the battery for one period using the *Execute* plan.

Observe that the first three plans already contain initial known (necessary) domain constraints for applicability, by means of conditions $\psi_X(\cdot, \cdot, \cdot)$. For instance, plan *SetCharge* may not be considered in a given instance if the module is only allowed to change charge directions once every four periods and charging in this period would violate this constraint. Similarly, plan *SetDischarge* may be ruled out because discharging the module implies that regardless of how the remaining modules are configured, the response is bound to fall short of the request. When none of the first three plans apply for a given module, then BDI failure recovery may be employed to backtrack up and select a different configuration path until all constraints are satisfied (or all options are exhausted).

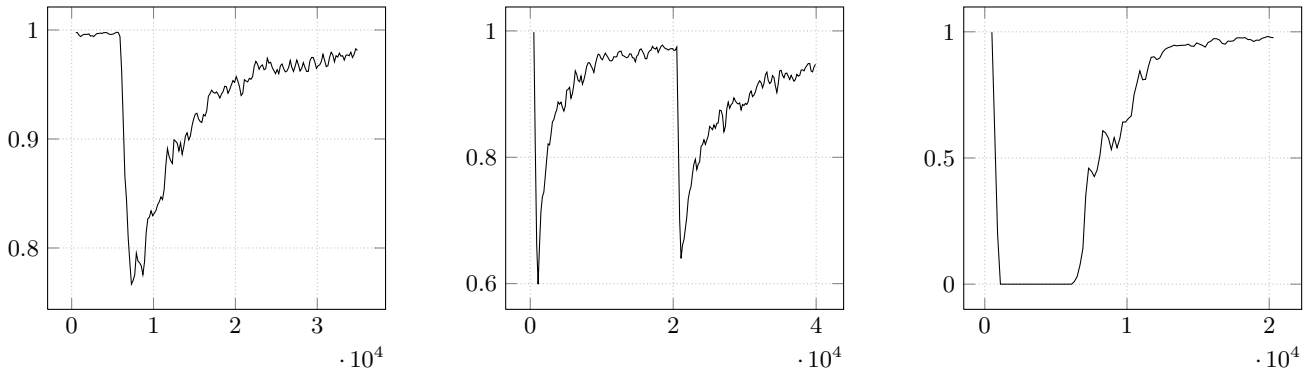
As such, the *Execute* plan is run only with configurations that are functionally correct. When this happens, the whole battery system is first executed for the period (action *operate*) and its performance evaluated via a sensor (action *evaluate*). If the evaluation step *succeeds*, then the desired overall rate r has been met and the whole execution is assumed successful. If, on the other hand, the desired overall rate has not been met in reality, the evaluation step *fails* and so does the whole execution of the program, since no BDI failure recovery can be used after the battery system has been physically operated.

5. EXPERIMENTAL RESULTS

In this section, we show experiments that demonstrate the suitability of the framework developed in Sections 2 and 3 using the energy domain example of Section 4. In particular, we report on three experiments that highlight the adaptive behaviour of the controller under different situations in which the environment dynamics are changing. The first experiment shows how the agent recovers functionality against standard deterioration in module capacities. In the second one, the agent is exposed to partial (temporal) failures of the system caused by modules malfunctioning, but which do not completely preclude the battery from successfully resolving system requests. Lastly, in the third experiment, we analyse the controller response and learning behaviour in the extreme case where the system suffers complete failure for some time and is thereafter restored.

Experimentation Setup

The following experimental setup applies to all experiments. We conducted experiments for a battery system with *five* modules (i.e., $k = 5$). For each module, the current charge state is described by a discrete value in the range $[0 : 3]$, where zero indicates a



(a) Capacity deterioration. (b) Partial failure with restoration. (c) Complete failure with restoration.

Figure 3: Controller performance in the energy application. Each plot shows success (y-axis) over the number of episodes (x-axis).

fully discharged state and three indicates a fully charged state. In addition, each module has an assigned configuration for the current period from the set $\{+c, 0, -c\}$, where $c = 1/k$. The operational model is simple: charging is meant to add $+c$ while discharging is meant to add $-c$ to a module's charge state, otherwise the state is unchanged for the period (i.e., there is no charge loss). Thus, the desired overall battery response is in the (normalized) range $[-1.0 : +1.0]$ in discrete intervals of $\pm c$. The complete state space for the problem is described by the number of modules (5), the possible requests (11), the charge state of the system (4^5), and the assigned configuration of the system (3^5), that is, $5 \times 11 \times 4^5 \times 3^5 \approx 13.7$ million states. Though significant, note that the agent does not have to learn over this space, because the filtering of nonsensical configurations by the plans' context conditions $\psi_X(\cdot, \cdot, \cdot)$ will reduce it substantially.

At the beginning of each learning episode, the configuration of every module is reset to 0 (i.e., not in use). The charge state of each module, however, is left untouched and carries over from the previous episode, as would be the case in the actual deployed system. This has implications for learning, particularly that the state space is *not* sampled uniformly. Each episode corresponds to one $G(r, 5, s)$ goal-event request: achieve overall battery response of r given current module charge states s. For simplicity of analysis, we assume only satisfiable requests: a solution always exists. The outcome of each episode is either no response (no configuration was executed), or a single invocation of the *Execute* plan for operating (and evaluating) the battery. The tolerance level is set to 0, so that the battery response is deemed successful only when the sum of the module configurations matches the request exactly.

In normal BDI operation, only plans that are applicable, as determined by their context condition, are considered for execution. For our learning framework, where applicability is additionally defined by a plan's decision tree, this means that only plans with a reasonable likelihood of success should be allowed. To represent this, we use an *applicability threshold* for plan selection of 40%, meaning that plans with a likelihood of success below this value are removed from consideration. While this feature does not alter the overall learning performance of the battery controller, it does preclude the battery from being operated (i.e., plan *Execute* being called) under module configurations that are likely to be unsuccessful. In fact, we found that the threshold used reduces the number of battery operations by 12%, which is substantial when considering battery life. Again, we stress that the difference in learning performance without the applicability threshold is not significant.

The threshold parameter for stability calculation is set to 0.5. We use an averaging window of $n = 5$ for both the stability-based metric $\mathcal{C}_s(\cdot, \cdot, n)$ and the world-based metric $\mathcal{C}_d(\cdot, n)$, and a (balanced) weighting of $\alpha = 0.5$ for the final confidence measure $\mathcal{C}(\cdot, \cdot, \cdot)$. Finally, each experiment is run five times and the reported results are averages from these runs.

Experiment 1: Capacity Deterioration

In this experiment we model the (typical) situation where module capacities deteriorate over time. In a real system this will happen gradually over several years of typical use. However, to show the response to substantial change, we force this deterioration to occur instantaneously in this experiment. Figure 3(a) shows the results for this case. In the beginning of the experiment, the system performs ideally as programmed, and goes about recording its experiences although there is no evident use of the resulting learning yet. After some time (about 5k episodes), the capacity of all five modules drops instantaneously, from the initial range $[0 : 3]$ to range $[0 : 2]$. These capacity changes result in a rapid drop in performance corresponding to the set of programmed/learnt solutions that no longer work. The ideally programmed system would, at this point, converge to $\approx 76\%$ performance. The adaptive system, however, aptly rectifies the situation by learning to avoid the module configurations that no longer work.

Experiment 2: Partial Failure with Restoration

In this scenario, we model a series of module malfunctions and their subsequent restoration. However, the battery always remains capable of successfully responding to the request using alternative configurations. In the experiment, the first battery module fails for the duration $[0 : 20\text{k}]$ after which it is reinstated, the second module fails for the period $[20\text{k} : 40\text{k}]$, and so on. Figure 3(b) shows the system performance for this stetting. At the beginning of each change, namely, at 0k and and 20k, the performance drops dramatically, as the expected solutions that utilise the failed module no longer work.[2] Following each module failure, the system successfully learns to operate the battery without it, by always configuring the failed module to not-in-use (i.e., state 0). Importantly, by the time each failed module is restored (e.g., iteration 20k for the first module), the system has already learnt to operate without it, and

[2]The apparent difference between the performance drop at 0k and 20k is not meaningful in any way; it just happens that more "bad" cases occurred in the first failure. The theoretical drop in performance is 45%.

hence, will not try to re-use it unless really required.

Experiment 3: Complete Failure with Restoration

In this experiment, we model the extreme scenario of complete failure of the system for some time followed by full restoration. Technically, *all* module configurations would fail for the period $[0 : 5\text{k}]$, after which they are restored to normal operation. The results are shown in Figure 3(c). At the beginning of the experiment, overall performance drops to zero rapidly, as none of the ideal configurations are successful in responding to the request. After a while (at around 2k episodes), the estimated likelihood of success of all plans drops below the applicability threshold of 40%. At this point, the battery operation comes to a complete halt: no plans are ever applicable and plan *Execute* is never invoked. Then, at episode 5k, the failed modules are repaired so that the battery is restored to normal operation. However, observe that since no plans are ever tried due to the applicability threshold, then new learning may not occur. To tackle this issue, we use as a "soft" applicability threshold mechanism: the 40% threshold applies 90% of the time. This allows the battery to operate with some likelihood and the agent system to eventually start recovering at around 6k episodes.[3]

In summary, the three types of scenarios described here for the energy storage domain empirically demonstrate the ability of a learning BDI agent to adapt to changes in the dynamics of the environment using the framework of Section 2 combined with the confidence metric developed in Section 3.

6. CONCLUSION AND DISCUSSION

The main contribution of this paper is a plan-selection learning mechanism for BDI agent-oriented programming languages and architectures that is able to *adapt* when the dynamics of the environment in which the agent is situated changes over time. Specifically, we proposed a *dynamic confidence measure* that combines ideas of plan stability [1] and plan coverage-based confidence [18, 19] from previous approaches, with a sense of the rate at which new worlds are being witnessed. This new confidence measure provides a simple way for the agent to judge how much it should trust its current understanding (of how well available plans can solve goal-events). In contrast with previous proposals, the confidence in a plan may *not* increase monotonically; indeed, it will drop whenever the learned behavior becomes less successful in the environment, thus allowing for new plan exploration to recover goal achievability. Importantly, however, the new measure subsumes previous approaches as it still preserves the traditional monotonic convergence expected in environments with fixed dynamics. Furthermore, the new mechanism does not require any account of the number of possible choices below a plan in the hierarchy, as is the case with the previous coverage-based approaches [18, 19], and hence scales up for any general goal-plan structure irrespective of its complexity. We demonstrated the effectiveness of the proposed BDI learning framework using a simplified energy storage domain whose dynamics is intrinsically changing: module performance within a battery system deteriorates over time or even completely fails on occasion on the one hand, while being periodically improved through system maintenance and upgrades on the other.

Apart from the already discussed approaches in [1, 18, 19] on which this work is based, the issue of combining learning with deliberation in BDI agent systems has not been widely addressed in the literature. Hernández et al. [9] reported preliminary results on learning the context condition for a single plan using a decision tree in a simple paint-world example, although they do not consider the issue of learning in plan hierarchies. In terms of approaches that integrate *offline* learning with deliberation in BDI systems, the work in [10] gives a detailed account using a real-world ship berthing logistics application. The authors take operational shipping data to train a neural network offline that is then integrated into the BDI deliberation cycle to improve plan selection. They show that the trained system is able to outperform the human operators in terms of scheduling the docking of ships to loading berths. Similar approaches integrating offline learning with BDI deliberation at runtime have previously also been used in robotic soccer [15, 5].

The work of [17] has highlighted the relationship between BDI and Markov Decision Processes on which the reinforcement learning literature is founded. Recently, Broekens et al. [4] reported progress on integrating reinforcement learning to improve plan selection in GOAL, a declarative agent programming language in the BDI flavour. They use an abstract state representation using only the stack of applicable rules and a sum cost heuristic that captures the number of pending goals. The intent is to keep the representation domain independent, with the focus on improving the plan selection functionality in the framework itself. In that way, their approach complements ours, and may be integrated as "meta-level" learning to influence the plan selection calculation given by the weighting function Ω (see Section 3). We note that such work is still preliminary and it is difficult to ascertain the generality of their approach in other domains. Nevertheless, their early results are encouraging in that the agent always achieves the goal state in less number of tries with learning enabled than without. Our work also relates to the existing work in hierarchical reinforcement learning [2], where task hierarchies similar to those of BDI programs are used. Of particular interest is the early work by Dietterich [7] that supports learning at all levels in the task hierarchy (as we do in our learning framework) in contrast to waiting for learning to converge at the bottom levels first.

There are several limitations and assumptions in the learning framework we have used. One issue, of course, has to do with maintaining the training set of past execution experiences per plan, indexed by world states. Simply storing such data may become infeasible after the agent has been operating for a long period of time. Importantly, the larger the training set, the more effort is required to induce the corresponding decision tree. For the latter problem, one option is to filter the training data at hand based on some heuristic, and only use a subset of the complete experience set to induce the decision tree. For instance, we experimented with filtering the training data based on the recency of the world states experienced. In our example energy domain we were able to reduce the size of the data set used in training by almost 75% by removing "old" experiences with no noticeable change in agent performance. The generality of such data-filtering heuristics, however, is unclear and requires further investigation to make any claims. Using *incremental* approaches for inducing decision trees [20, 21] will certainly address both problems, but may impact classification accuracy.

Perhaps a more severe limitation of the learning framework proposed is that it cannot account for interactions between a plan's subgoals. For instance, consider a travel-agent system that has two subgoals to book a flight and hotel accommodation on a fixed budget. Indeed, the way a flight is booked will impact the funds re-

[3]Note that the battery will be actually operated after five module configuration plans *Set** have been selected and carried out (one per existing module). In the best case, only one of these plans has failed the threshold and hence there is a 10% chance that the battery will operate. On the other hand, if all plans are failing the applicability threshold, then there is only a 0.1^5 (i.e., 0.00001) chance that the battery will operate. In a real system, the agent should have some input from the environment indicating that some important changes have ocurred (e.g., some batteries have been replaced/repaired).

maining for the next hotel booking goal, and some flight options may leave the agent unable to book any hotel at all. Since our agents have no information of the higher "agenda" at the subgoal level, there is no way for such dependencies to be learnt. One way of addressing this limitation may be to consider extended notions of execution traces that take into account *all* subgoals that led to the final outcome, but this requires further work.

We have reported important improvements to a framework for learning plan selection in BDI agent systems. While there is still work to do to resolve issues around long-term learning and interactions between subgoals, we believe that the framework has matured to the point of integration into practical systems. We have demonstrated this in the design of a battery controller based on a real-world scenario.

7. REFERENCES

[1] S. Airiau, L. Padgham, S. Sardina, and S. Sen. Enhancing the adaptation of BDI agents using learning techniques. *International Journal of Agent Technologies and Systems (IJATS)*, 1(2):1–18, Jan. 2009.

[2] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[3] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley, 2007.

[4] J. Broekens, K. Hindriks, and P. Wiggers. Reinforcement learning as heuristic for action-rule preferences. In *Proceedings of the Programming Multiagent Systems Languages, Frameworks, Techniques and Tools workshop (PROMAS)*, pages 85–100, 2010.

[5] J. Brusey. *Learning Behaviours for Robot Soccer*. PhD thesis, RMIT University, 2002.

[6] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5, Jan. 1999. Agent Oriented Software Pty. Ltd.

[7] T. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[8] M. P. Georgeff and F. F. Ingrand. Decision making in an embedded reasoning system. In *Proceedings of IJCAI*, pages 972–978, Detroit, USA, 1989.

[9] A. Guerra-Hernández, A. E. Fallah-Seghrouchni, and H. Soldano. *Learning in BDI Multi-agent Systems*, volume 3259 of *LNCS*, pages 218–233. Springer, 2004.

[10] P. Lokuge and D. Alahakoon. Improving the adaptability in automated vessel scheduling in container ports using intelligent software agents. *European Journal of Operational Research*, 177, March 2007.

[11] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[12] B. Norris, G. Ball, P. Lex, and V. Scaini. Grid-connected solar energy storage using the zinc-bromine flow battery. In *Proceedings of the Solar Conference*, pages 119–122. American Solar Energy Society, 2002.

[13] A. Pokahr, L. Braubach, and W. Lamersdorf. JADEX: Implementing a BDI-infrastructure for JADE agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, Sept. 2003.

[14] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[15] M. Riedmiller, A. Merke, D. Meier, A. Hoffman, A. Sinner, O. Thate, and R. Ehrmann. Karlsruhe brainstormers - a reinforcement learning approach to robotic soccer. In *RoboCup 2000: Robot Soccer World Cup IV*, 2001.

[16] P. H. Shaw, B. Farwer, and R. H. Bordini. Theoretical and experimental results on the goal-plan tree problem. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1379–1382, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[17] G. I. Simari and S. Parsons. On the relationship between mdps and the bdi architecture. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS'06)*, pages 1041–1048, New York, NY, USA, 2006. ACM.

[18] D. Singh, S. Sardina, and L. Padgham. Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems (RAS)*, 58:1067–1075, 2010.

[19] D. Singh, S. Sardina, L. Padgham, and S. Airiau. Learning context conditions for BDI plan selection. In van der Hoek, Kaminka, Lespérance, Luck, and Sen, editors, *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 325–332, Toronto, Canada, May 2010.

[20] E. Swere, D. Mulvaney, and I. Sillitoe. A fast memory-efficient incremental decision tree algorithm and its application to mobile robot navigation. In *Proc. of IROS*, 2006.

[21] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

[22] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X, http://www.csc.liv.ac.uk/∼mjw/pubs/imas/.