Integrating Learning into a BDI Agent for Environments with Changing Dynamics

Dhirendra Singh, Sebastian Sardina, Lin Padgham
School of Computer Science and IT
RMIT University, Melbourne, Australia
{dhirendra.singh,sebastian.sardina,lin.padgham}@rmit.edu.au

Geoff James
CSIRO Energy Technology
Sydney, Australia
geoff.james@csiro.au

Abstract

We propose a framework that adds learning for improving plan selection in the popular BDI agent programming paradigm. In contrast with previous proposals, the approach given here is able to scale up well with the complexity of the agent's plan library. Technically, we develop a novel confidence measure which allows the agent to adjust its reliance on the learning dynamically, facilitating in principle infinitely many (re)learning phases. We demonstrate the benefits of the approach in an example controller for energy management.

1 Introduction

The Belief-Desire-Intentions (BDI) agent programming paradigm [Wooldridge, 2009; Busetta *et al.*, 1999; Pokahr *et al.*, 2003; Bordini *et al.*, 2007] is a successful and popular approach to developing complex systems that behave robustly in situations where the environment is interacting with the system. However, BDI agents do not incorporate learning, and once deployed, have no ability to adjust to changes in the dynamics of an environment that cause previously successful approaches to fail and vice versa. In this work we maintain the structure and advantages of a BDI agent program, but integrate adaptive machine learning techniques that are generally used to learn behaviour in a fixed environment (often off-line with a training set), for use in online continuous learning in an environment where what works, may well change over time. In that way, our contribution is to agent programming—*to make BDI programs more adaptive by seamlessly integrating learning techniques*—rather that to machine learning research. The aim is to use the operational knowledge encoded in the agent program, but learn refinements to it so as to cope with changes in the dynamics of the environment.

A key issue when learning is that of exploitation *vs.* exploration: how much to trust and exploit the current (learnt) knowledge, versus how much to try things in order to gain new knowledge. In our earlier work [Singh *et al.*, 2010a; 2010b], a "coverage-based" measure of confidence was used to capture how much the BDI agent should trust its current understanding (of good plan selection), and therefore exploit rather than explore. Intuitively, this confidence was based on the degree to which the space of possible execution options for the plan has been explored (covered). The more this space has been explored, the greater the confidence, and the more likely the agent is to exploit. As recognised in this work, 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 a smart building equipped with a large battery system that can be charged when there is excess power, and used (discharged) when there is excess demand, in order to achieve an overall desired building consumption rate for some period. A battery installation is built from independent modules with different chemistries. Initially, the embedded battery controller can be programmed (or can learn) to operate optimally. However, over time modules may operate less well or even cease to function altogether. In addition, modules may be replaced with some frequency. Thus, what is needed is a controller that, after having explored the space well and developed high confidence in its learning, is able to observe when learned knowledge becomes unstable, and dynamically modify its confidence allowing for new exploration and revised learning in an ongoing way. To achieve this we develop a confidence measure based on an assessment of how well informed our selections are, combined with an observation of the extent to which we are seeing new situations.

The rest of the paper is as follows. First, we introduce BDI systems and machine learning concepts necessary to understand this work. We then describe our BDI learning framework, initially developed in [Airiau *et al.*, 2009; Singh *et al.*, 2010a; 2010b], and extend this with our new confidence measure (Section 3.2) for learning in environments with changing dynamics. Next, we describe a battery controller agent, based on a real application that requires ongoing learning, and show how it adjusts in a variety of ways to an environment where battery behaviour changes. We conclude with related work and limitations requiring future work.

2 Preliminaries

2.1 BDI Agent Systems

BDI agent programming languages are typically built around an explicit representation of propositional attitudes (e.g., beliefs, desires, intentions, etc.). A BDI architecture addresses how these components are modelled, 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 [Busetta *et al.*, 1999], JADEX [Pokahr *et al.*, 2003], and Jason [Bordini *et al.*, 2007] among others. Specifically, a BDI intelligent agent systematically chooses and executes *plans* (i.e., operational procedures) to achieve or realise 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 such, 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.

By grouping together plans responding to the same event type, the agent's plan library may be viewed as a set of goal-plan tree templates (e.g., Figure 1): 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 interact directly 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 world states).

Figure 1 shows the possible outcomes when plan P is selected in a given world w. In order for the first subgoal G_1 to succeed, plan P_A must be selected followed by P_I (successes marked as $\surd$). The *active execution trace*[1] for this selection is described as $\lambda_1 = G[P : w] \cdot G_1[P_A : w] \cdot G_3[P_I : w]$ (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 in world w. Subsequently subgoal G_2 is posted whose successful resolution is given 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 w' in λ_2 is the resulting world from the successful execution

[1]An active execution trace [Singh *et al.*, 2010b] describes decision sequences that terminate in the execution of a leaf plan.

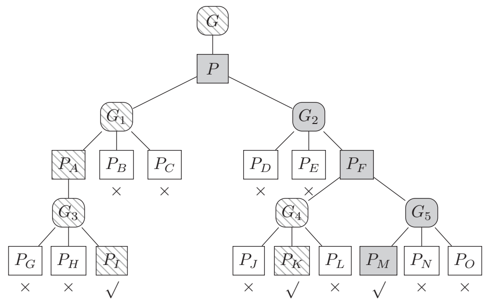


Figure 1: An example goal-plan hierarchy.

of leaf plan P_I in the preceding trace λ_1. Similarly, w'' is the resulting world from the execution of P_K in λ_2. There is only one way for plan P to succeed in the initial world w, namely, that described by the three traces λ_1, λ_2 and λ_3. All other execution traces lead to failures (marked as $\times$).

As can be seen, appropriate plan selection is crucial in BDI systems. Whereas standard BDI platforms leverage domain expertise by means of *fixed* logical context conditions of plans, here we are interested in ways of improving or *learning* plan selection for a situated agent, based on experience.

2.2 Machine Learning Concepts

Our aim is to determine a plan's applicability (context) condition by learning the structure of a *decision tree* [Mitchell, 1997] from experiences of trying the plan in different situations. The idea is that the learnt structure will summarise the set of experiences (i.e., plan failed in world w_1, passed in w_2, etc.) into a compact form, i.e., correctly *classify* them (as pass or fail), and may be used to decide the likely outcome in unseen situations. We use decision trees because *(i)* they support hypotheses that are a disjunction of conjunctive terms and this is compatible with how context formulas are generally written; *(ii)* they can be converted to human readable rules and validated by an expert; and *(iii)* they are robust against "noisy" data such as generally successful plans nevertheless failing due to unobservable factors. Specifically, we use the algorithm J48 (a version of C4.5) from the weka learning package [Witten and Frank, 1999].

The second concept important for this work is that of *confidence*: how much should the agent rely on its current learning, given that it must act while learning. Confidence determines the exploration strategy: the lower it is, the less reliable the learning is and the more the agent *explores* the options available to it; the higher it is, the more it *exploits* its (learnt) knowledge. Typically (e.g., in reinforcement learning) the exploration strategy is fixed upfront (constant or monotonically decreasing, e.g., in ϵ-greedy and Boltzmann exploration [Sutton and Barto, 1998]). The basic assumption is that learning is a one-off process that improves over time. We argue, however, that for an embedded learning agent this assumption is too limiting. Instead, we assume that in the course of its lifetime the deployed agent will likely experience many changes in the environment: some of which will cause previously learnt solutions to no longer work. For this reason, we propose a dynamic confidence measure (Section 3.2) that *adapts* accordingly with changes in performance, and forms the basis for a more practical heuristic for guiding exploration.

3 The Integrated BDI Learning Framework

We now describe our BDI learning framework [Airiau *et al.*, 2009; Singh *et al.*, 2010a; 2010b], which is a seamless integration of standard Belief-Desire-Intention (BDI) agent-oriented programming [Wooldridge, 2009] with decision tree learning [Mitchell, 1997]. Within the framework, three mechanisms are crucial. First is a principled approach to learning context conditions based on execution experiences (Section 3.1). Second is a dynamic confidence measure in ongoing learning to decide how much trust to put in the current understanding of the world (Section 3.2). Finally, an adequate plan selection scheme compatible with the new type of plans' preconditions is required (Section 3.3).

3.1 Learning Context Conditions from Experience

As discussed in our earlier work [Airiau *et al.*, 2009; Singh *et al.*, 2010a; 2010b], plans' context conditions are first generalised to decision trees [Mitchell, 1997], which can be learnt over time. 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 in 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* in the course of trying plans in various world states.

When it comes to the learning process, the training set for a given plan's decision tree contains samples of the form $[w, e, o]$, where w is the world state—a vector of discrete attributes—in which the plan was executed, e is the vector of parameters for the goal-event that this plan handles, 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 for different event-goal parameter values 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 and event-goal parameters that are relevant to the plan's (real) context condition.

The typical *offline* use of decision trees assumes availability of the complete training set. In our case, however, learning and acting are interleaved in an *online* manner: the agent uses its current learning to make plan choices, gaining new knowledge that impacts subsequent choices. This raises the issue of deciding how "good" the intermediate generalisations are. We have previously addressed this issue in two different ways. The first [Airiau *et al.*, 2009] is to be selective in recording experiences based on how sensible the related decisions were. Referring to Figure 1, consider the case where plan selection results in the failed execution trace $\lambda_3' = G[P : w] \cdot G_2[P_F : w'] \cdot G_5[P_N : w'']$, after traces λ_1 and λ_2 have been successfully carried out. In this case, if a negative outcome is recorded for P_F and P, this is misleading. These non-leaf plans failed not because they were a bad choice for worlds w' and w respectively but because a bad choice (P_N for G_5 in w'') was made further down in the hierarchy. To resolve this issue, we used a *stability* filter in [Airiau *et al.*, 2009] to record failures only for those plans whose outcomes are considered to be stable, or "well-informed." The second approach is to record always, but instead adjust a plan's selection probability based on some measure of *confidence* in its decision tree. In [Singh *et al.*, 2010a; 2010b], we considered 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 currently: the more choices that have been explored, the greater the confidence in the resulting decision tree.

The approach of using confidence to adjust plan selection weights is more generally applicable to learning in BDI hierarchies, as we showed in [Singh *et al.*, 2010a; 2010b], and is the one we use in this study. However, the traditional assumption that learning is a one-off process is no longer applicable for embedded learning agents that operate over long time frames. These must in fact often un-learn and re-learn as necessary, due to ongoing changes in the environment dynamics. The measure of confidence—that effectively defines the exploration strategy—is therefore no longer a monotonically increasing function, and must dynamically adjust accordingly if previously learnt solutions no longer work. Our new confidence measure for such environments with changing dynamics is presented next. Importantly, this measure subsumes the functionality of the former methods, in that it has equivalent convergence properties when used in environments with fixed dynamics.

3.2 Determining Confidence in Ongoing Learning

We now describe our confidence measure for learning in environments with continually changing dynamics and where the agent must invariably "keep on learning." It uses the notion of *stability* that we introduced in [Airiau *et al.*, 2009], and defined in [Singh *et al.*, 2010b] as follows: *"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."* Here the success rate is the percentage of successful executions in the last few experiences in w. Stability is then calculated by looking at the difference in consecutive rates. Our aim is to use this notion to judge how 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 $\lambda = G_1[P_1 : w_1] \cdots G_\ell[P_\ell : w_\ell]$ we define

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

where $\Delta_{app}(G_i, w_i)$ denotes the set of all applicable plans (i.e., whose boolean context conditions hold true) in world w_i for goal event G_i, and $stable(P, w_i)$ holds true if plan P is deemed stable in world w_i, as defined in [Singh *et al.*, 2010b].

For instance, take the failed execution trace $\lambda'_3 = G[P : w] \cdot G_2[P_F : w'] \cdot G_5[P_N : w'']$ from before and suppose that the applicable plans are $\Delta_{app}(G, w) = \{P\}$, $\Delta_{app}(G_2, w') = \{P_D, P_F\}$, and $\Delta_{app}(G_5, w'') = \{P_M, P_N, P_O\}$. Say also that P_D and P_N are the only plans deemed stable (in worlds w' and w'', respectively). Then the degree of stability for the whole trace is $\zeta(\lambda'_3) = 2/6$. Similarly, for the two subtraces $\lambda_a = G_2[P_F : w'] \cdot G_5[P_N : w'']$ and $\lambda_b = G_5[P_N : w'']$ of λ'_3, we get $\zeta(\lambda_a) = 2/5$ and $\zeta(\lambda_b) = 1/3$.

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 plan P we store degree $\zeta(\lambda_a)$ whereas for plan P_F we record degree $\zeta(\lambda_b)$. Leaf plans, like P_N, make no choices so their degree is assigned 1. Intuitively, by doing this, we record against each plan in the (failed) trace, an estimate of how informed the choices made "below" the plan were. Algorithm 1 describes how this (hierarchical) recording happens given an active execution trace λ. Here *RecordDegreeStability*(P, w, d) appends degree d for plan P in world state w to the list of experiences.

Algorithm 1: *RecordDegreeStabilityInTrace*(λ)

```
Data: $\lambda = G_1[P_1 : w_1] \cdot \ldots \cdot G_\ell[P_\ell : w_\ell]$, with $\ell \geq 1$.
Result: Records degree of stability for plans in $\lambda$.
if ($\ell > 1$) then
    $\lambda' = G_2[P_2 : w_2] \cdot \ldots \cdot G_\ell[P_\ell : w_\ell]$;
    RecordDegreeStability($P_1, w_1, \zeta(\lambda')$);
    RecordDegreeStabilityInTrace($\lambda'$);
else
    RecordDegreeStability($P_1, w_1, 1$);
```

As 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 degree 1 against it. In other words, choices that led to success are considered well-informed, but for failures we want to know just how good the decisions were. The fact that all plans do eventually become stable or succeed, means that $\zeta(\lambda)$ is guaranteed to converge to 1 (for fixed dynamics).

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 taken when performing P in w were over the n most recent executions. Notice that if the environment dynamics are constant, this measure gradually 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 dynamics of the environment were to change and (working) plans start to fail or become unstable, then the measure of confidence adjusts accordingly.

Normally, generalising using decision trees is useful, if one has "enough" data. For us, this amounts to trying a plan in a given world in meaningful ways (captured by $\mathcal{C}_s(P, w, n)$), as well as in different worlds. To measure the latter, we monitor *the rate at which new worlds are being witnessed by a plan* P. During early exploration, most worlds that a plan is selected for will be unique, thus yielding a high rate (corresponding to low confidence). Over time, the plan would get selected in all worlds in which it is reachable and the rate of new worlds would approach zero (corresponding to full confidence). Given this, we define our second confidence metric $\mathcal{C}_d(P, n) = 1 - \frac{|NewStates(P,n)|}{n}$, where *NewStates*$(P, n)$ is the set of worlds in the last n executions of P that have not been witnessed before. $\mathcal{C}_d$ normally converges to 1.0 after all worlds where the plan might be considered are eventually witnessed.

We now define our final (aggregated) confidence measure $\mathcal{C}(P, w, n)$ using the stability-based metric $\mathcal{C}_s(P, w, n)$ that reflects how well-informed the last n executions of plan P in world w were, and the domain metric $\mathcal{C}_d(P, n)$ that captures how well-known the worlds in the last n executions of plan P were compared with what we had experienced before:

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

where α is a weighting factor used to set a preference bias between the two component metrics. Here n controls the sensitivity to performance changes: smaller values make the measure more responsive; larger values yield greater stability.[2]

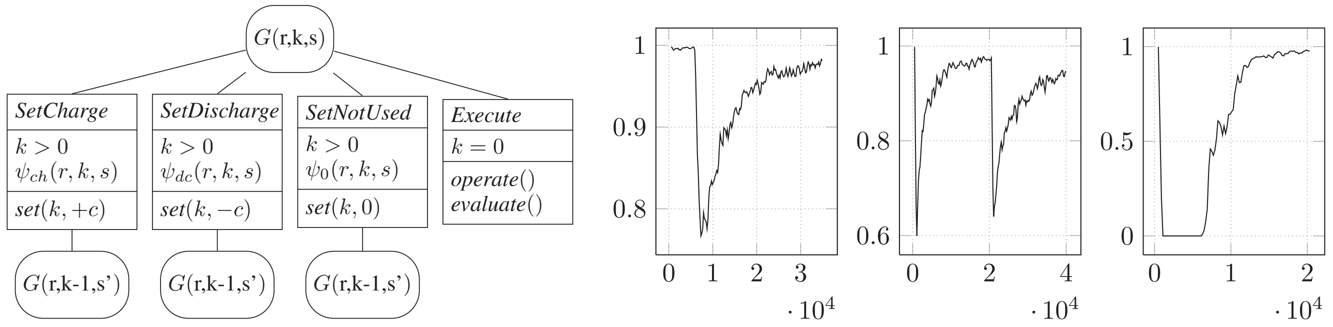
3.3 Using Learning and Confidence to Select Plans

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. To do this, we make the agent choose probabilistically from the set of 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. We define this selection weight for plan P in world w relative to 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 $\mathcal{P}(P, w)$ stands for the probability of success of plan P in world w as given by P's decision tree. This weight definition is similar to what we have used before [Singh *et al.*, 2010a; 2010b], except for the new confidence metric $\mathcal{C}$ defined above. The idea is to combine the likelihood of success of plan P in world w—term $\mathcal{P}(P, w)$—with a confidence bias (here $\mathcal{C}(\cdot, \cdot, \cdot) \in [0, 1]$) to determine a final plan selection weight Ω. When the confidence is maximum (i.e., 1.0), then $\Omega = \mathcal{P}(P, w)$, and the final weight equals the likelihood reported by the decision tree; when the confidence is zero, then $\Omega = 0.5$, and the decision tree has no bearing on the final weight (a default weight of 0.5 is used instead). Intuitively, Ω describes the exploration strategy: when confidence is low, the probability of choosing the believed high success option is also low, leading to a strong chance of exploring alternatives.

[2]Currently n is decided by the programmer. This could be lifted by using, say, a temporally reduced weighting for past executions.



(a) Goal-plan hierarchy for a k-modules battery system. (b) Capacity deterioration. (c) Partial failure. (d) Complete failure.

Figure 2: The implemented BDI controller and its (success) performance (y-axis) over the number of episodes (x-axis).

4 An Embedded Battery System Controller

Large battery systems enable increasing levels of renewable energy in our electricity system. Such installations usually consist of multiple modules that can be controlled independently [Norris *et al.*, 2002]. Since battery performance is susceptible to changes over time (e.g., modules may lose actual capacity) an *adaptable* control mechanism is desirable that accounts for such changes. Here, we describe an *embedded BDI controller for a modular battery system*, that regulates overall energy consumption of a smart office building comprise of loads (e.g., appliances) and generators (e.g., solar panels), by suitably ordering the battery to charge (i.e., act as a load) or discharge (i.e., act as a generator) at determined rates (the overall rate being the sum over the modules' rates).

Figure 2(a) shows our implemented controller for an overall system capacity of c (module capacity) $\times$ k_{max} (number of modules). Goal-event $G(r, k, s)$ requests a normalized battery response rate of $r \in [-1, 1]$, where -1 (1) indicates maximum discharge (charge) rate, for a current module k and battery state s—initially, k is set to k_{max}. The controller works by recursively configuring each module (i.e., from k_{max} down to 1) using plans *SetCharge* (charge at rate $+c$), *SetDischarge* (discharge at rate $-c$), and *SetNotUsed* (disconnect, i.e., rate 0), and thereafter (i.e., when $k = 0$) physically operating the battery for one period using the *Execute* plan.

Observe that the first three plans contain known domain constraints for applicability using condition ψ_X. For instance, plan *SetCharge* may not apply if the module is already charged; *SetDischarge* may be ruled out if discharging the module means that no matter how the remaining modules are configured, the response will fall short of the request. When none of the plans apply, then BDI failure recovery is employed to backtrack and select a different configuration path until all constraints are satisfied or all options exhausted.

Plan *Execute* is therefore run only with functionally correct configurations. It first operates the whole battery for the period (*operate*) and then evaluates the result via a sensor (*evaluate*). If the evaluation *succeeds*, then the desired rate r has been met and the whole execution is deemed successful. Otherwise, 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. Over time, the system learns the real applicability conditions of plans by training over their observed outcomes in different situations. In this way, programmed conditions act as an initial filter followed by learning to decide final applicability.

4.1 Experimental Results

All experiments used a battery system with five modules (i.e., $k_{max} = 5$). Each module had a charge state in the range 0 (fully discharged) to 3 (fully charged), as well as an assigned configuration for the period from $\{+c, 0, -c\}$, where $c = 1/k_{max}$. Charging adds $+c$ while discharging adds $-c$ to a module's charge state, otherwise the state is unchanged for the period (i.e., there is no charge loss). Thus, the net battery response is in the (normalized) range $[-1, 1]$ in discrete intervals of $\pm c$. The state space for the problem is given by the modules (5), the requests (11), the charge states (4^5), and the assigned configurations (3^5), that is, $5 \times 11 \times 4^5 \times 3^5 \approx 13.7$ million. The agent does not learn over the full space, however, since the filtering of nonsensical configurations by the plans' context conditions $\psi_X(\cdot, \cdot, \cdot)$ reduces it substantially (to ≈ 1.5 million). 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 generate only satisfiable requests. The threshold for stability calculation is set to 0.5. We use an averaging window of $n = 5$ for both the stability-based metric $\mathcal{C}_s$ and domain metric $\mathcal{C}_d$, and a (balanced) weighting of $\alpha = 0.5$ for the final confidence measure $\mathcal{C}$.[3] Each experiment is run five times and the reported results are averages from these runs. Finally, since in normal BDI operation only plans deemed applicable as per their context condition are considered for execution, then for our learning framework we used an *applicability threshold* of 40%, meaning that plans with a selection weight below this value are not considered.[4]

The first experiment (Figure 2(b)) represents the traditional (one-off) learning setting. A deterioration in module capacities (at $\approx$ 5k episodes) causes a rapid drop in performance (to $\approx 76\%$) corresponding to the set of programmed/learnt solutions that no longer work. Performance is subsequently rectified by learning to avoid configurations that no longer work.

[3]Lower stability threshold gives greater accuracy (more samples) and vice versa. α can be tuned towards $\mathcal{C}_d$ when the world is relatively complex compared to the structure of choices, or $\mathcal{C}_s$ otherwise.

[4]The threshold does not alter overall performance, but does prevent battery use when it is unlikely to succeed. We found that it reduces operations by 12%, which is significant for battery life.

The next two experiments demonstrate adaptive behaviour when the environment dynamics is continually changing. In Figure 2(c), the agent is exposed to partial failures from modules malfunctioning (the first module fails during $[0, 20\text{k}]$ and is then restored, the second during $[20\text{k}, 40\text{k}]$, and so on), but the battery remains capable of responding to requests using alternative configurations. The agent successfully learns to operate the battery without the failing module each time. Finally, in Figure 2(d)) the controller adapts in the extreme case of complete failure (during $[0, 5\text{k}]$).[5] The key point in all experiments is that the learner does not require explicit notification of changes: it re-learns by adjusting exploration based on its performance (that reflects the impact of such changes).

5 Conclusion

We proposed 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, the proposed *dynamic 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 to previous proposals, the confidence in a plan may *not* always steadily, and it will drop whenever the learned behavior becomes less successful in the environment, thus allowing for new plan exploration to recover goal achievability. The new learning mechanism subsumes previous approaches in that it still preserves the traditional monotonic convergence in environments with fixed dynamics. Furthermore, unlike in [Singh *et al.*, 2010a; 2010b], it does not require any account of the number of possible choices below a plan in the hierarchy, 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.

Perhaps the most severe limitation of our learning framework is that it cannot account for interactions between a plan's subgoals. If a plan involves two sub-goals (e.g., book flight and book hotel), these are learnt independently of each other and this "local" learning may yield sub-optimal behavior. One way of addressing this may be to use extended notions of execution traces that take into account *all* subgoals that led to the final execution outcome.

The issue of combining learning within BDI deliberation has not been widely addressed in the literature. Guerra-Hernández *et al.* [2004] reported preliminary results on learning the context condition for a *single* plan, and hence do not consider the issue of learning in plan hierarchies. Other work has focused on integrating offline (rather than online) learning within deliberation in BDI systems, such as the work of Lokuge and Alahakoon [2007] for ship berthing logistics and that of Riedmiller *et al.* [2001] within a robotic soccer domain. The recent approach of Broekens *et al.* [2010] that integrates reinforcement learning to improve rule selection in the GOAL agent language complements ours, and may be used as "meta-level" learning to influence the plan selection weight Ω. Indeed, we plan to investigate this in future work.

[5]Around 2k episodes, the selection weight of *all* plans drops below the applicability threshold. To ensure that learning can still progress, we use as a "soft" applicability threshold mechanism: the 40% threshold applies 90% of the time.

Acknowledgments

We thank the anonymous reviewers for their comments. We also acknowledge the support of Agent Oriented Software and the Australian Research Council (grant LP0882234).

References

[Airiau *et al.*, 2009] Stéphane Airiau, Lin Padgham, Sebastian Sardina, and Sandip Sen. Enhancing the adaptation of BDI agents using learning techniques. *International Journal of Agent Technologies and Systems (IJATS)*, 1(2):1–18, January 2009.

[Bordini *et al.*, 2007] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[Broekens *et al.*, 2010] Joost Broekens, Koen Hindriks, and Pascal Wiggers. Reinforcement learning as heuristic for action-rule preferences. In *Proceedings of PROMAS*, pages 85–100, 2010.

[Busetta *et al.*, 1999] Paolo Busetta, Ralph Rönnquist, Andew Hodgson, and Andrew Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5, January 1999. Agent Oriented Software Pty. Ltd.

[Guerra-Hernández *et al.*, 2004] Alejandro Guerra-Hernández, Amal El Fallah-Seghrouchni, and Henry Soldano. *Learning in BDI Multi-agent Systems*, volume 3259 of *LNCS*, pages 218–233. Springer, 2004.

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

[Mitchell, 1997] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

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

[Pokahr *et al.*, 2003] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. JADEX: Implementing a BDI-infrastructure for JADE agents.*EXP: In search of innovation*, 3(3):76–85, 2003.

[Riedmiller *et al.*, 2001] 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.

[Singh *et al.*, 2010a] Dhirendra Singh, Sebastian Sardina, and Lin Padgham. Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems (RAS)*, 58:1067–1075, 2010.

[Singh *et al.*, 2010b] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Stephane Airiau. Learning context conditions for BDI plan selection. In *Proc. of AAMAS*, pages 325–332, 2010.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Witten and Frank, 1999] Ian Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[Wooldridge, 2009] Michael Wooldridge. *Introduction to Multi-Agent Systems*. Second edition, 2009.