Adaptive Cognitive Agents: Updating Action Descriptions and Plans

Peter Stringer[1 [0000−0002−8760−4717]], Rafael C. Cardoso[2 [0000−0001−6666−6954]], Clare Dixon[1 [0000−0002−4610−9533]], Michael Fisher[1 [0000−0002−0875−3862]], and Louise A. Dennis[1 [0000−0003−1426−1896]]

[1] The University of Manchester, Manchester, UK
peter.stringer, clare.dixon, michael.fisher, louise.dennis@manchester.ac.uk
[2] University of Aberdeen, Aberdeen, UK
rafael.cardoso@abdn.ac.uk

Abstract. In this paper we present an extension of Belief-Desire-Intention agents which can adapt their performance in response to changes in their environment. We consider situations in which the agent's actions no longer perform as anticipated. Our agents maintain explicit descriptions of the expected behaviour of their actions, are able to track action performance, learn new action descriptions and patch affected plans at runtime. Our main contributions are the underlying theoretical mechanisms for data collection about action performance, the synthesis of new action descriptions from this data and the integration with plan reconfiguration. The mechanisms are supported by a practical implementation to validate the approach.

Keywords: Beliefs-Desires-Intentions, Action Descriptions, AI Planning

1 Introduction

Long-term autonomy requires autonomous systems to adapt once their capabilities no longer perform as expected. To achieve this, a system must first be capable of detecting such changes and then adapting its internal reasoning processes to accommodate these. For example, deploying an autonomous robot into a dynamic environment can result in actions becoming unreliable over time, as the environment changes, producing unexpected outcomes that were unforeseeable before runtime. The autonomous agent must be capable of observing these changes and adapting accordingly.

Cognitive agents [6, 29, 35] have explicit reasons for the choices they make. These are often described in terms of the agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. This view of cognitive agents is encapsulated within the Belief-Desire-Intention (BDI) model [28, 29]. Here, *beliefs* represent the agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment, *desires* represent the agent's long-term goals,

while *intentions* represent the goals that the agent is actively pursuing (the representation of intentions often includes partially instantiated and/or executed plans and so combines the goal with its intended means).

Our work focuses on cognitive agents programmed in a Belief-Desire-Intention (BDI) [30] programming language providing high-level decision making in an autonomous system, as outlined in [15]. Programs written in these languages use *plans* created in advance by a programmer to select *actions* to execute in the environment. These plans make implicit assumptions about the behaviour of the actions they execute. Therefore, in this context, the challenge becomes to make these assumptions explicit, detect when they no longer hold, and then modify the plans accordingly. Most agent programming languages commonly used for high-level control of autonomous robots, do not support the adaptation of agent programs at runtime to deal with changes in their environment.

Some of these languages use *action descriptions* (sometimes referred to as *capabilities* in the literature), which consist of explicit pre- and post-conditions for all known actions. An action's pre-conditions are the environment conditions which should hold if an action is to execute correctly, whilst post-conditions are the expected changes in the environment made as a direct result of the completed action. We assume the existence of these action descriptions. We also assume that the cognitive agent is able to determine: when an action has finished executing; and whether it has met its post-conditions when it does so. These assumptions allow the system to maintain logs of action performance which can then be mined to detect patterns of failure. Although not all BDI systems can represent action descriptions, some do, and so mechanisms and semantics used for such functionality are discussed in [23, 12, 33].

Once a failure pattern is detected, we use synthesis methods to update its action description to reflect its actual behaviour. We can then repair or replace actions in any existing plans by using an automated planner to construct patches.

Running Example Consider an agent navigating around a space to visit some set of waypoints (where, for instance, it needs to perform some kind of inspection tasks). Examples of this kind are common (see [27, 19]). We assume the agent has a predetermined route to traverse the waypoints — for instance that the robot should visit waypoint 0, then 1, then 2, then 3 before returning to 0. It also has actions that encode movement between waypoints (e.g., `move(0, 1)` moves the robot from point 0 to point 1). As well as the specific actions needed for the predetermined route, the agent is also aware that it can move between the other waypoints (for instance that it can move from point 0, directly to point 3, `move(0, 3)` and from point 3 to point 1, `move(3, 1)`). If, over time, the route between points 0 and 1 becomes obstructed, we would like the robot to reason that it can replace the instruction to move from 0 to 1 directly, in its plan, with an instruction to move from 0 to 3, and then from 3 to 1.

Our contribution, in this work, is a methodology to detect faulty actions, modify their descriptions and reconfigure BDI plans based on these new descriptions, enabling long term autonomy. Our work applies in general to BDI programming languages which allow action descriptions. We have implemented

the methodology in the GWENDOLEN programming language as a prototype to exemplify the approach.

2 Background and Related Work

GWENDOLEN is a BDI programming language that contains a number of features for integrating with autonomous and robotic systems. One of its main distinctive features is that GWENDOLEN agents can be verified using a program model-checker, Agent Java Pathfinder (AJPF) [16]. A full operational semantics for GWENDOLEN is presented in [13]. Its key components are, for each agent, a set of beliefs that are ground first-order formulae and a set of intentions that are stacks of *deeds* associated with some event. Deeds can be the addition or deletion of beliefs, the adoption of new goals, and the execution of primitive actions. A GWENDOLEN agent may have several concurrent intentions and will, by default, execute the first deed on each intention stack in turn. GWENDOLEN is event-driven and events include the acquisition of new beliefs (typically via perception), messages and goals. A programmer supplies plans that describe how an agent should react to events by extending the deed stack of the relevant intention. These plans contain actions for execution.

Plans are of the form `event : guard <- deeds`, where the event is the addition or deletion of a belief or goal, the guard is a term that is evaluated against the agent's belief set and the deeds are transformed into an intention stack if the event occurs and the guard evaluates to true.

If implemented in GWENDOLEN, our example of a robot travelling between waypoints can be represented with the four plans shown in Figure 1. We use

```
+!at(1): {B at(0)} <- move(0, 1), +!at(2);
+!at(2): {B at(1)} <- move(1, 2), +!at(3);
+!at(3): {B at(2)} <- move(2, 3), +!at(0);
+!at(0): {B at(3)} <- move(3, 0), +!at(1);
```

Fig. 1. Four GWENDOLEN plans for a patrolling robot.

standard BDI syntax in which `!` represents a goal and + denotes the addition of this goal. The first of these plans states that if a new goal to be at waypoint 1 has been added (`+!at(1)`) and the agent currently believes it is at waypoint 0 (`B at(0)`) then the agent should move to waypoint 1 (`move(0, 1)`) and adopt the goal to be at waypoint 2 (`+!at(2)`) to continue its patrol route. For example, if the robot starts at waypoint 0 and is sent a goal to reach waypoint 1, then these four plans will keep the robot patrolling around all four waypoints autonomously.

While all BDI languages have individual features, they have many similarities. In particular the use of plans (sometimes called *rules*) which have guards controlling when they apply and then execute some sequence of actions, belief updates and goal updates is common to many such languages (e.g., *Jason* [3] and GOAL [23]).

Some BDI languages also employ *action descriptions* (sometimes referred to as *capabilities*) which have their roots in AI planning and STRIPS operators [18].

Definition 1 (Action Description). *We assume a language $\mathcal{L}$ of first-order terms.* Action descriptions *are a triple $\{Pre\}A\{Post\}$ where A is a term in $\mathcal{L}$ representing an action, $\{Pre\}$ is a set of terms representing the action's pre-conditions and $\{Post\}$ is a set of expressions of the form $+t$ or $-t$ (where t is a term in $\mathcal{L}$). Note: $+t$ means that the term t should be added to the agent's belief base following execution of the action and $-t$ means that the term t should be removed from the agent's belief base.*

Returning to our example, the action description $\{at(0)\}move(0,1)\{-at(0),+at(1)\}$ can be associated with the agent action `move(0, 1)`. This has the pre-condition, $\{at(0)\}$ (the agent is at waypoint 0), and post-conditions $\{-at(0),+at(1)\}$ (the belief that the agent is at waypoint 0 should be removed and the belief that the agent is at waypoint 1 should be added).

In many languages, actions descriptions are used both to control whether an action is executed if it appears in a plan (by checking the action's pre-conditions) and to directly manipulate the agent's belief base using the post-conditions without using perception to check whether the action has completed successfully and established these post-conditions. In some cases it is implicitly assumed that the low-level action execution process checks post-conditions and so a success signal is not sent to the agent unless the post-conditions have been achieved.

Action descriptions/capabilities exist in, among others, the GOAL [24] language and 2APL [11]. A version of GWENDOLEN also exists that contains an implementation of action descriptions [33].

GWENDOLEN does not use its action descriptions to control action execution or to update its belief base. Instead it uses the descriptions to make inferences about action success or failure by comparing the state of the world after an action execution completes with the state of the world described in the post-conditions. This allows the agent to react to action failure as well as, more generally, to plan failure. GWENDOLEN also tracks the performance of actions over time in an *action log*. An example of an action log using the `move(0, 1)` action is shown in Figure 2. This shows a log with two entries. Each entry contains the action name, a list of the difference in beliefs before and after the action executed, and finally the outcome for that action once it terminated. The action in Figure 2 is a move action from waypoint 0 to waypoint 1. In the first entry, the action is believed to have succeeded and the change in beliefs is shown as the addition of the belief $at(1)$ (at waypoint 1) and removal of the belief $at(0)$ which matches the expected post-conditions for that action. In the second entry, the change in beliefs results in the agent believing that it is at waypoint 3, not at waypoint 1 as per the action description, producing a failure as the action outcome.

The action log has a fixed, application specific size, and the oldest entry is removed before adding a new one, once the log reaches its size limit. The presence of this action log opens up the possibility of implementing an *action*

Action	Change in Beliefs	Action Outcome
move(0, 1)	+at(1), -at(0)	Success
move(0, 1)	+at(3),-at(0)	Failure

Fig. 2. Example of an action log with the move(0, 1) action for a GWENDOLEN agent.

lifecycle [34] inspired by the concept of goal life-cycles for BDI languages [22]. An action lifecycle allows actions which fail or are aborted to be moved into a *suspect* state and finally become *deprecated* following repeated failures.

We're not aware of any other BDI language that maintains an action log in this way, but in principle, it should not be difficult to add this functionality to any language that already supports action descriptions.

The automated planning research community has invested considerable effort in the modelling of actions with stochastic outcomes, both theoretically as variants on Markov Decision Procedures [26, 36], and practically by capturing such concepts in planners (e.g. [9]) and domain description languages such as in the Planning Domain Definition Language (PDDL) 2.1 [20]. This community deploys action descriptions to flexibly plan on-the-fly for each new goal which avoids the problem faced in BDI languages that an action whose behaviour has changed may result in failing, and therefore useless, plans. The use of a BDI language, with its programmer supplied plans, presumes that bespoke planning for every new goal is undesirable (usually for reasons of efficiency, but also for verifiability). Our approach exploits AI planning techniques to patch the plans that fail as a result of action failure, but seeks to minimize the amount of planning that actually takes place.

Plan failure has been extensively researched in BDI programming languages (e.g., [4, 31, 17]), however, it has not been linked with action descriptions perhaps because most languages do not use action descriptions as a mechanism to detect action failure. The closest work to our own is in [22] with a proposal for BDI goal life-cycles.

A key component of our approach is synthesising or learning a new action description when an action ceases to perform as expected. We presume this arises because of the dynamic environment in which the agent is operating. Using algorithms to discover the effects of actions has been explored in the AI Planning domain [2]. Most of the resulting techniques operate in environments where it is assumed multiple action descriptions need to be learned at the same time and that the action descriptions themselves have not been changing during the learning period. We have based our approach on ideas from [10] and [21] in which new action descriptions are learned from traces of action behaviour with a weighting mechanism used to guide choice of additions and deletions to the constructed action post-condition.

After learning/synthesising a new (or updated) action description, it is necessary to refactor the plans from the plan library. The process of updating plans of execution based on a set of conditions (failure or new information) is often

referred to as reconfigurability, and it has been frequently applied in the robotics and manufacturing industry [8, 32, 5, 1]. A mechanism for plan library reconfigurability combining BDI agents and automated planning was presented in [7], but it has no account for how failure is detected, and simply ignores the action that caused the failure in subsequent reconfigurations. We leverage this work in ours, if an action is deprecated by the action lifecycle, then any plans involving that action are patched using the mechanism from [7].

To the best of our knowledge, there is no end-to-end framework in cognitive agents for updating action descriptions and patching the associated plans such as is presented here.

3 Framework

Our starting point is the system architecture outlined in [15] in which a cognitive agent performs high-level mission reasoning, such as deciding in which order some set of waypoints are to be visited. In order to do this, it takes input from subsystems for processing sensor data into high-level concepts such as the location of obstacles, and outputs instructions (actions) to control systems such as those for navigation and path planning. This is shown in Figure 3 together with the action log component that tracks action performance.

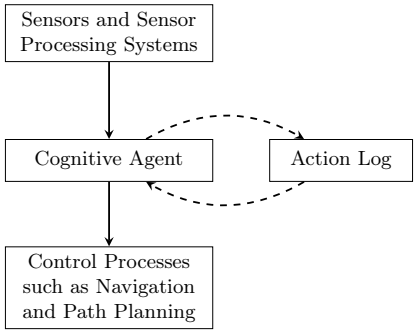

Fig. 3. System architecture overview.

Cognitive agents generally employ a *reasoning cycle* which governs a sense-reason-act process. The action log integrates with the *act* phase and compares the outcomes of executed actions to the post-conditions described in the action's description. If the post-conditions are successful, then a success is logged, and if they are not a failure is logged. In all situations the action log also records the changes in beliefs from the moment when the action was executed, to the moment when it succeeded or failed. These changes are stored as a list of expressions of the form $+t$ or $-t$ where t is a term — that is, in the same format as post-conditions in action descriptions.

Figure 4 shows five entries in an action log. When a new entry is made by an agent following an action execution, the size of the log is checked against the

predefined size limit. If the size limit has been reached, then the oldest entry is removed (from the top of the list in this example) before the new entry is added to the bottom. In this case a single action success has been experienced for the `move(0, 1)` action, followed by four failures. These failures might be caused by, for instance, some obstacle appearing in the path between waypoints 0 and 1. Attempts by the agent to move around the obstacle, using low-level obstacle avoidance techniques have led consistently to the agent finding itself at waypoint 3, each time this action is executed in the route goal.

The agent is able to navigate the rest of the route in this example but finds that the `move(0, 1)` action leads the agent to believe they have reached waypoint 3, and when executing this action during the next four iterations of the route, the same outcome is recorded.

Our framework extends action descriptions to include a *failure threshold* (Definition 2).

Definition 2 (Action Description (modified)). Action descriptions *are a tuple* $\{Pre\}A\{Post\}[n]$ *where* A, $\{Pre\}$ *and* $\{Post\}$ *are as described in Definition 1 and* n *is a positive integer representing a failure threshold.*

If the number of failures for the action in the action log exceeds the failure threshold, then the action becomes deprecated. Note that the action log should be of fixed length, so that an action can not become deprecated as the result of a slow build up of occasional failure over time. It only becomes deprecated if its *recent* failures have exceeded the threshold. The definition of recent should be application specific to account for speed with which change/degradation is anticipated in the environment. The threshold should be specific to the action itself, since some actions are naturally more failure prone than others for reasons that may be external to the action itself. Our tolerance of failure therefore varies depending upon the action.

We extend the act phase of the reasoning so that after the execution of an action, the action log is consulted. If the most recent action has not become deprecated the cycle continues as before. If it has become deprecated, then a new action description is synthesised from the information in the log and relevant

Action	Change in Beliefs	Action Outcome
move(0, 1)	+at(1),-at(0)	Success
move(0, 1)	+at(3),-at(0)	Failure
move(0, 1)	+at(3),-at(0)	Failure
move(0, 1)	+at(3),-at(0)	Failure
move(0, 1)	+at(3),-at(0)	Failure

Fig. 4. Example of detecting failures in an action log with size limit equal to 5. The next new entry will be added to the bottom row and the first row would be removed.

plans are patched before the agent continues to the sense phase. This reasoning cycle is shown in Figure 5.

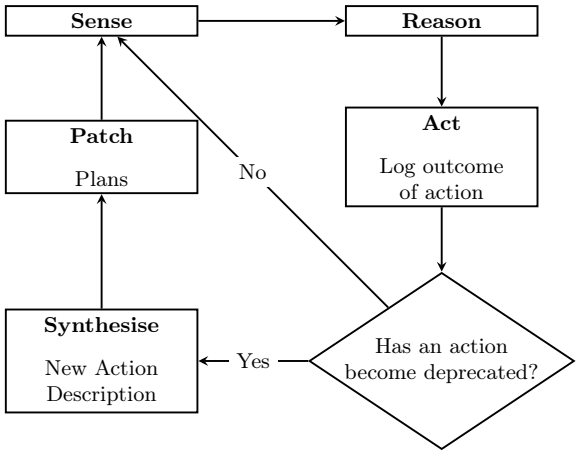

Fig. 5. Extended Sense-Reason-Act cycle to account for action deprecation, synthesis of new action descriptions, and the patching of plans.

We synthesise a new action description by extracting, from the action log, all the failed instances of the deprecated action. We then have a list (probably containing duplicates, as can be observed from Figure 4) of new candidate post-conditions for the action in the form of the change in beliefs as the action executed. Each item in this list is assigned a weight score based on how recent the item is. The weights for identical items are then summed and that with the highest score selected as the new post-condition for the action. Pseudo-code for this process is shown in Algorithm 1. Line 2 instantiates the initial weight score (n) to 1, and in Line 3 it sets *post_scores* to an empty map. Lines 4–7 will loop through every entry in the action log to find entries that match with the deprecated action (same action) and where the outcome of the entry was reported as a failure. When this happens, the post-conditions of the action are added to the *post_scores* map along with the weight score, which is then incremented by one for the future iterations of the action log. In line 8 we initialise *best* with 0. Lines 9–11 iterate over the keys in the *post_scores* map to select the candidate post-condition with the highest weight score.

If we consider the action log in Figure 4 and suppose our failure threshold is four, then the agent's 'act' phase should now attempt to synthesise a new action description from the log. It extracts the list of failures which contains four items all of which have identical new post-conditions — namely $\{+at(3), -at(0)\}$. This therefore becomes the new post-condition for the action `move(0, 1)`.

However, suppose the action log is more variable. Initially, attempts to avoid the obstacle between 0 and 1 resulted in the agent arriving at waypoint 3, but suppose the obstacle has become more serious — perhaps sand and debris is

Algorithm 1: Algorithm for synthesising post-conditions when an action is detected to be deprecated.

```
1  if action is deprecated then
2      n ← 1;
3      post_scores ← {} // map of post-conditions to scores
4      for entry ∈ action log do
           // NB. the action log consists of tuples (action, change in
              beliefs, outcome)
5          if entry[0] = action & entry[2] = Failure then
6              post_scores[entry[1]] ← post_scores[entry[1]] + n;
7              n ← n + 1
8      best ← 0;
9      for post ∈ keys(post_scores) do
10         if post_scores[post] > best then
11             best ← post
```

piling up as the result of storms — and now the low-level movement behaviour causes an abort that returns the agent to waypoint 0. This results in the action log in Figure 6.

Action	Change in Beliefs	Action Outcome
move(0, 1)	+at(1),-at(0)	Success
move(0, 1)	+at(3),-at(0)	Failure
move(0, 1)	+at(3),-at(0)	Failure
move(0, 1)		Failure
move(0, 1)		Failure

Fig. 6. Example of an action log with variable post-conditions for the same action (move(0, 1)).

Figure 7 shows this action log extracted into a list of candidate post-conditions, weighted by how recent they are.

Of the two candidate post-conditions $\{+at(3), -at(0)\}$ has a total weight of 3, while {} (no change) has a total weight of 7. Therefore the empty post-condition is selected for the new action description.

Once a new action description is stored, we are able to use a plan reconfiguration mechanism to patch any plans containing the action. The work in [7] describes how an AI planning problem can be extracted from a failed action by a process of:

Candidate Post-Condition	Weight
$\{+at(3), -at(0)\}$	1
$\{+at(3), -at(0)\}$	2
$\{\}$	3
$\{\}$	4

Fig. 7. Post-conditions extracted from Figure 6, added with their respective weights which are calculated based on how recent they are.

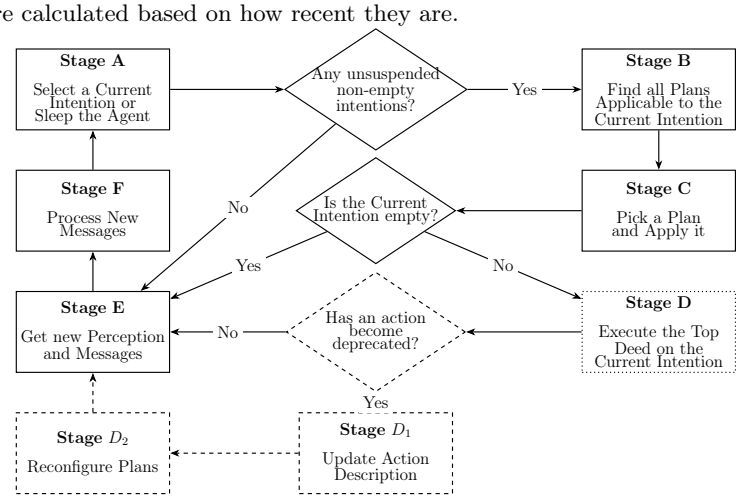

Fig. 8. GWENDOLEN reasoning cycle. Our additions are shown with dashed lines and stages we have modified are shown with dotted lines.

1. using the failed action's pre- and post-conditions as initial and goal states respectively for the planning problem; and
2. using the set of all other action descriptions as an action model for the planner.

This planning problem can then be solved to create a "patch" for any BDI plan containing the failed action. Our framework uses this mechanism with a slight modification. We only seek to replace an action once it has become deprecated (i.e., after some pre-defined number of failures). The set of action descriptions sent to the planner is then created from the agent's current set of action descriptions, including the newly learned description of the deprecated action.

In our example, let us assume that our `move(0,1)` action has become deprecated. Attempts to move from waypoint 0 to waypoint 1 now result in the agent arriving at waypoint 3 (based on the action log from Figure 4. A STRIPS-type planner [18] is called with the updated action descriptions and an initial planning state — `at(0)` (the agent is at waypoint 0) — and goal state — `at(1) & ¬ at(0)` (the agent should end up at waypoint 1) — created from the pre- and post-conditions of `move(0, 1)`. Among other action descriptions the planner has the new description for `move(0,1)` available ($\{at(0)\}$`move(0, 1)`$\{-at(0), +at(3)\}[4]$ with [4] representing the failure threshold of the action) as well as an action de-

scribing a move from waypoint 3 to 1 ($\{at(3)\}$`move(3, 1)`$\{-at(3), +at(1)\}[4]$). It is straightforward for the planner to create the plan `move(0,1),move(3,1)` to solve this problem (note that `move(0,1)` now takes us to waypoint 3, not waypoint 1). If we were using plans similar to the GWENDOLEN plans[3] shown in Figure 1, this means that the plan `+!at(1):{at(0)} <- move(0, 1), +!at(2)` contains our deprecated action and will not succeed in moving the agent to waypoint 1. This patch produced by the planner, replaces the appearance of `move(0, 1)` in the original plan producing the new plan: `+!at(1):{at(0)} <- move(0, 1), move(3, 1), +!at(2)` which is stored for reuse.

4 Implementation

We implemented our framework in the version of the GWENDOLEN programming language that creates an action log of action success and failure using action descriptions [33].[4]

We extended the GWENDOLEN reasoning cycle with a synthesise stage (Stage D_1) and a reconfigure stage (Stage D_2) which are executed after GWENDOLEN's equivalent of the act phase which is called Stage D. This reconfigure stage uses the action log to synthesise new action descriptions and then uses these to patch the agent's plans. Our extended GWENDOLEN reasoning cycle is shown in Figure 8 with our additions shown using dashes.

After Stage D (when actions are executed), the last entry of the action log is checked. If it is an entry for anything other than an action failure nothing further happens, no action becomes deprecated, and the cycle continues to Stage E. However, if it is an entry for an action that has failed, the number of entries containing a failure for this specific action is checked against its failure threshold. Note that the threshold value of an action is domain specific. If the threshold has been reached, the reasoning cycle moves to the new Stage D_1 in which a new action description will be learned and then to Stage D_2 where plans will be patched.

A fixed length action log may not capture rare, but still consistent, failures, as the oldest entry is removed when new entries are recorded. A more sophisticated failure threshold could be developed to measure the significance of each failure regardless of frequency and act accordingly, although this case was not considered in the current implementation.

There is also scope for further development to allow greater refinement of failure thresholds, which is not limited to just failure frequency. In the current state, the punishment for assigning an inappropriate threshold is not considerable, as agents would quickly reach the threshold again to correct the action description back to the original description. This is the current state of managing incorrect failure detections. However, this method is wholly reliant upon accurate agent perception of the environment and actions could produce further failures if the

[3] As noted, many BDI formalisms represent plans in a very similar fashion, so although we use a GWENDOLEN plan as an example here, the technique is general.

[4] Code available at https://github.com/mcapl/mcapl/tree/reconfig_eumas

agent wrongly believes pre-conditions for actions. This system works under the assumption that the agent's perception of the environment remains accurate. Further testing and deployment into a realistic scenario would be required to improve on the current implementation.

Once a new action description is stored we are able to use plan reconfiguration mechanism from [7]. This extracts all the action descriptions from the agent and translates them into STRIPS operators [18]. Let, $\{Pre\}a\{Post\}[n]$ be the old action description for the failed action a. The reconfiguration mechanism computes initial and goal states for a planning problem from $\{Pre\}$ and $\{Post\}$. This planning problem is then given to a STRIPS planner together with the STRIPS operators of the agent's plan descriptions. If the planner computes a new plan this is translated into a sequence of GWENDOLEN actions, a_l, this sequence replaces a everywhere it appears in the agent's plans.

5 Evaluation

We evaluated our approach on a variation of the "waypoint patrol" example we have been using throughout the paper. Our environment consisted of five waypoints and our agent had a plan for a patrol mission to visit each waypoint in turn. The GWENDOLEN plan was:

```
+!at(4):{at(0)} <- move(0, 1),
                   move(1, 2),
                   move(2, 3),
                   move(3, 4);
```

Each move action had a description of the form:

$$\{at(X)\}\texttt{move(X, Y)}\{-at(X), +at(Y)\}$$

(e.g., $\{at(1)\}$`move(1, 2)`$\{-at(1), +at(2)\}$. We varied the number of action descriptions for 'move' actions available to the agent. The agent always had descriptions for the four actions in the plan (i.e., `move(0, 1)`, `move(1, 2)`, `move(2, 3)`, `move(3, 4)` — we refer to these as the *fixed actions*), but also had a random selection of other 'move' actions between the five waypoints — we refer to these as the *variable actions*. Figure 9 illustrates this, with the *fixed* move actions shown by solid lines and the *variable* move actions shown by dashed lines.

We generated random instances of this scenario varying the probability that each of the variable 'move' actions was available. The table presented in Figure 10 shows how many times (out of ten runs) our framework successfully managed to patch the plan in the event that the `move(0, 1)` action resulted in the agent finding itself at waypoint 2 rather than waypoint 1.

As is to be expected, we can see that as the number of potential alternative actions increases, so does the chance of successfully patching the failing plan. In particular, once more than 50% of the edges in the graph are available as actions, there is a high chance that the agent will be able to synthesise a patch for its plans.

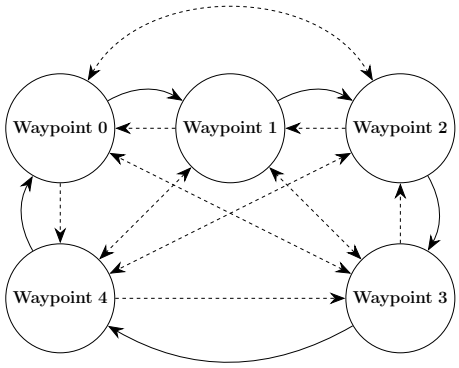

Fig. 9. Waypoint environment. Dashed arrows indicate variable actions only available in *some* instantiations of the problem.

10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
3	7	7	8	9	10	9	10	10	10

Fig. 10. First row represents the probability for each extra move route to be available in the execution. Second row contains the results for how many times our implementation managed to successfully patch a plan when the action `move(0, 1)` action was deprecated resulting in a move to waypoint 2, rather than waypoint 1

When there was only a 10% probability of each *variable action* being available, the reconfigured plan, when it could be generated, tended to be quite long. For instance, in one instance, the only *variable action* available at runtime was the `move(3, 1)` action. This resulted in a patch were `move(0, 1)` was replaced by the sequence `move(0, 1), move(2, 3), move(3, 1)` (recall that `move(0, 1)` is now resulting in a move to waypoint 2). This resulted in the patched plan:

```
+!at(4):{at(0)} <- move(0, 1),
                   move(2, 3),
                   move(3, 1),
                   move(1, 2),
                   move(2, 3),
                   move(3, 4);
```

The shortest possible plan patch, can be achieved for when the `move(2, 1)` action is available to the planner. When possible, the planner always opts for the plan with the lowest "cost" that can achieve the provided goal state. We modelled the plan cost simply as the total number of actions in the plan. In our scenario this provides a good estimation for the lowest cost (in terms of resources consumed by the agent in order to execute the plan) since all of the actions are similar, though this would not necessarily hold true for other action models. This costing approach explains why we tended to generate shorter patches when more actions were available.

6 Discussion and Future Work

One major aspect of future work is to adapt the framework to manage action descriptions containing variables. Many BDI languages use variables and unification in plans, to enable one plan to apply in many situations depending upon the instantiation of its parameters. There are two aspects to this challenge. Firstly when an action is executed in a BDI language, it is almost always the case that its variable parameters are instantiated — so although we might have an action description of the form $\{at(X)\}$`move(X, Y)`$\{-at(X), +at(Y)\}$ where X and Y are variables, it is only ever called as, say `move(0, 1)` or `move(1, 2)`. Therefore the process of synthesising new descriptions from the action log will need to utilize generalisation techniques to abstract from concrete log entries to abstract descriptions. It may also be necessary to split action descriptions by synthesising new pre-conditions indicating that, in some situations the action still behaves as originally assumed, but in others it does not. Secondly, STRIPS-type planners, while they frequently use action descriptions that contain variable parameters, do not generally plan using initial and goal states that contain variables. This includes the planner embedded in the implementation we used from [7] — therefore this planner would need to be replaced with one capable of handling variables in initial states and goals. The work in [25] contains simple examples that might be adapted for this use.

We would also like to introduce more sophistication into the algorithm for learning new action descriptions. At present all changes in beliefs after an action execution are treated as one group. Consider a situation where two robots are both working in an area. Sometimes, after moving between waypoints the agent also perceives the presence of the second robot. In this case the current action log would sometimes record $\{+at(1), -at(0), +second\_robot\}$ as the belief change and sometimes record $\{+at(1), -at(0)\}$. Algorithm 1 treats these entirely separately and is unable to recognise that $+at(1)$ and $-at(0)$ occur in both. We anticipate that weighting each term appearing in the set of belief changes individually, rather than as a group, would enable the construction of post-conditions that better reflected the actual action behaviour.

At present the planning problem sent to the STRIPS planner is formulated from the description of the failed action alone and does not account for the context in which the action appears. Many BDI plans are expressed in terms of some *guard*, which can be considered a pre-condition for the whole plan, and a *goal* which can be considered a goal state for the plan. We would like to use techniques such as regression planning to infer from the plan's guard and goal, and the pre-conditions and post-conditions of any other actions in the plan, what the actual state of the agent is likely to be at the point the failed action was executed and which of the failed action's post-conditions were actually necessary in order to achieve the goal of the plan. This introduces more flexibility into the patching mechanism, allowing plans to be patched even if an exact replacement for the failing action could not be found. It also reduces the risk that the computed patch might contain additional post-conditions that will break the plan — for instance, suppose our failed action is $\{pr_1\}a_1\{+po_2\}$ and the computed

patch is a_2, a_3 where a_3's post-condition is $\{+po_1, +po_3\}$. Now consider a plan `e:guard <-` a_1`,`a_4 where the description of a_4 is $\{\neg po_3\}a_4\{+po_4\}$. If we replace a_1 in this plan with our patch then a_4 will no longer be applicable and the plan will break. More context-sensitive construction of the planning problem should be able to account for this and avoid creating a patch that will break the plan.

The use of the Gwendolen language which is linked to the AJPF model-checking tool and the Model-Checking Agent Programming Languages (MCAPL) framework [14], opens the possibility of verifying the patched plans produced by our framework. While we are interested in exploring this idea, the AJPF model-checker typically performs verification very slowly. If the agent existed in an environment where there were periods of inactivity, then it would be possible for re-verification to take place to ensure that the agent's plans continued to adhere to any specified properties, but in an environment where patching needs to occur quickly then this may not be feasible. If the reconfiguration mechanism was adapted, as suggested above, to be sensitive to the context in which an action was invoked then it should be possible to establish idealised results about the safety of patches, at least in environments where the only things changing the environment are the agent's own actions. It might also be possible to treat actions appearing in plans as sequences of abstract actions of length up to l, with the abstract actions having no specified behaviour during verification. This forces the verification to consider all possible action outcomes, allowing plans to be patched with any sequence of actions of length less than l, but the resulting state space for verification is likely to be unwieldy and include consideration of many action outcomes that are either unlikely or impossible, forcing, in turn, the inclusion of fail-safe plans within the agent to handle behaviour that can never occur resulting in "crufty" code.

The extent to which long-term autonomy can be achieved through the generation of amended action descriptions and the patching of plans is an open question. Scenarios such as we have presented involving navigation around waypoints linked in a graph structure are relatively common, and it is reasonable to suppose that over time paths between waypoints might alter. What is unknown is how common it is that changes in the environment or robot capabilities can be compensated for by combinations of (adapted) actions and how common it is that action degradation simply results in a robot that can not usefully perform its mission. It is likely that the proposed framework would need to be combined with mechanisms for weakening mission specifications, for instance, by dropping some goals that were no longer obtainable, while continuing to pursue others.

We have presented here the over-arching template of a framework for adapting BDI agent plans in the face of changed action behaviour. With the additions from the future work described, the framework in this paper should be capable of handling larger scenarios with greater complexity. Also, introducing multiple agents to the scenario undoubtedly increases the complexity of the situation, although this also opens the opportunity for agents to collaborate and share action descriptions.

Acknowledgements This work has been supported by The University of Manchester's Department of Computer Science and the EPSRC "Robotics and AI for Nuclear" (EP/R026084/1), "Future AI and Robotics for Space" (EP/R026092/1), and Computational Agent Responsibility (EP/W01081X/1) Hubs and the TAS Verifiability Node (EP/V026801). During the course of this work, Michael Fisher was supported by the Royal Academy of Engineering.

References

1. Antzoulatos, N., Castro, E., de Silva, L., Rocha, A.D., Ratchev, S., Barata, J.: A Multi-agent Framework for Capability-based Reconfiguration of Industrial Assembly Systems. International Journal of Production Research **55**(10), 2950–2960 (2017)
2. Arora, A., Fiorino, H., Pellier, D., Etivier, M., Pesty, S.: A review of learning planning action models. Knowledge Engineering Review **33** (2018)
3. Bordini, R.H., ubner, J.F.H., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason (2007)
4. Bordini, R.H., Hübner, J.F.: Semantics for the Jason Variant of AgentSpeak (Plan Failure and some Internal Actions). In: ECAI. pp. 635–640. IOS Press (2010). https://doi.org/10.3233/978-1-60750-606-5-635
5. Borgo, S., Cesta, A., Orlandini, A., Umbrico, A.: A planning-based architecture for a reconfigurable manufacturing system. In: Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling. pp. 358–366. ICAPS'16, AAAI Press, London, UK (2016)
6. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
7. Cardoso, R.C., Dennis, L.A., Fisher, M.: Plan library reconfigurability in BDI agents. In: Proc. of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS). p. 195–212. Springer (2019)
8. Chen, I.M., Yang, G., Yeo, S.H.: Automatic modeling for modular reconfigurable robotic systems: Theory and practice. In: Cubero, S. (ed.) Industrial Robotics, chap. 2. IntechOpen, Rijeka (2006)
9. Cirillo, M., Karlsson, L., Saffiotti, A.: Human-Aware Task-Planning: An Application to Mobile Robots. ACM Trans. Intelligent Systems Technology **1**(2), 15 (2010)
10. Cohen, P.R., Feigenbaum, E.A.: The handbook of artificial intelligence: Volume 3, vol. 3. Butterworth-Heinemann (2014)
11. Dastani, M.: 2APL: a practical agent programming language. Auton Agent Multi-Agent Syst (16), 214—-248 (2008). https://doi.org/https://doi.org/10.1007/s10458-008-9036-y
12. Dastani, M., van Birna Riemsdijk, M., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 39–67. Springer US, Boston, MA (2005). https://doi.org/10.1007/0-387-26350-0_2
13. Dennis, L.A.: Gwendolen semantics: 2017. Tech. Rep. ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
14. Dennis, L.A.: The mcapl framework including the agent infrastructure layer and agent java pathfinder. The Journal of Open Source Software **3**(24) (2018)

15. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical verification of decision-making in agent-based autonomous systems. Automated Software Engineering **23**(3), 305–359 (2016)
16. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Autom. Softw. Eng. **19**(1), 5–63 (2012). https://doi.org/10.1007/s10515-011-0088-x, https://doi.org/10.1007/s10515-011-0088-x
17. Ferrando, A., Cardoso, R.C.: Safety shields, an automated failure handling mechanism for bdi agents. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems. p. 1589–1591. AAMAS '22, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2022), https://www.ifaamas.org/Proceedings/aamas2022/pdfs/p1589.pdf
18. Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3), 189–208 (1971). https://doi.org/https://doi.org/10.1016/0004-3702(71)90010-5, https://www.sciencedirect.com/science/article/pii/0004370271900105
19. Fisher, M., Cardoso, R.C., Collins, E.C., Dadswell, C., Dennis, L.A., Dixon, C., Farrell, M., Ferrando, A., Huang, X., Jump, M., Kourtis, G., Lisitsa, A., Luckcuck, M., Luo, S., Page, V., Papacchini, F., Webster, M.: An overview of verification and validation challenges for inspection robots. Robotics **10**(2) (2021). https://doi.org/10.3390/robotics10020067, https://www.mdpi.com/2218-6581/10/2/67
20. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. JAIR **20**, 61–124 (2003)
21. Guerra-Hernández, A., Fallah-Seghrouchni, A.E., Soldano, H.: Learning in bdi multi-agent systems. In: International Workshop on Computational Logic in Multi-Agent Systems. pp. 218–233. Springer (2004)
22. Harland, J., Morley, D.N., Thangarajah, J., Yorke-Smith, N.: An operational semantics for the goal life-cycle in BDI agents. Autonomous agents and multi-agent systems **28**(4), 682–719 (2014). https://doi.org/10.1007/s10458-013-9238-9
23. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-89299-3_4
24. Hindriks, K.V.: Programming cognitive agents in goal (2021)
25. Luger, G.F.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Addison-Wesley Publishing Company, USA, 6th edn. (2008)
26. Mausam, Weld, D.S.: Planning with Durative Actions in Stochastic Domains. JAIR **31**, 33–82 (2008)
27. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification Patterns for Robotic Missions. IEEE Transactions on Software Engineering **47**(10), 2208–2224 (2021). https://doi.org/10.1109/TSE.2019.2945329, https://doi.org/10.1109/TSE.2019.2945329
28. Rao, A.S., Georgeff, M.P.: Modeling Agents within a BDI-Architecture. In: Proc. 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR&R). pp. 473–484. Morgan Kaufmann (1991)
29. Rao, A.S., Georgeff, M.P.: An Abstract Architecture for Rational Agents. In: Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR&R). pp. 439–449. Morgan Kaufmann (1992)
30. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. KR **92**, 439–449 (1992)

31. Sardina, S., Padgham, L.: A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning. Autonomous Agents and Multi-Agent Systems **23**(1), 18–70 (2011)
32. Støy, K., Brandt, D., Christensen, D.J.: Self-Reconfigurable Robots. MIT Press (2010)
33. Stringer, P., Cardoso, R.C., Dixon, C., Dennis, L.A.: Implementing durative actions with failure detection in gwendolen. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 332–351. Springer International Publishing, Cham (2022)
34. Stringer, P., Cardoso, R.C., Huang, X., Dennis, L.A.: Adaptable and Verifiable BDI Reasoning. In: Cardoso, R.C., Ferrando, A., Briola, D., Menghi, C., Ahlbrecht, T. (eds.) Proceedings of the First Workshop on Agents and Robots for reliable Engineered Autonomy, Virtual event, 4th September 2020. Electronic Proceedings in Theoretical Computer Science, vol. 319, pp. 117–125. Open Publishing Association (2020). https://doi.org/10.4204/EPTCS.319.9
35. Wooldridge, M., Rao, A. (eds.): Foundations of Rational Agency. Applied Logic Series, Kluwer Academic Publishers (1999)
36. Younes, H.L.A., Simmons, R.G.: Solving Generalized Semi-Markov Decision Processes using Continuous Phase-type Distributions. In: Proc. AAAI. pp. 742–747. AAAI Press (2004)