Plan Library Reconfigurability in BDI Agents*

Rafael C. Cardoso[0000−0001−6666−6954], Louise A. Dennis[0000−0003−1426−1896], and Michael Fisher[0000−0002−0875−3862]

University of Liverpool, Liverpool L69 3BX, United Kingdom
{rafael.cardoso,L.A.Dennis,mfisher}@liverpool.ac.uk

Abstract. One of the major advantages of modular architectures in robotic systems is the ability to add or replace nodes, without needing to rearrange the whole system. In this type of system, autonomous agents can aid in the decision making and high-level control of the robot. However, when autonomously replacing a node it can be difficult to reconfigure plans in the agent's plan library while retaining correctness. In this paper, we exploit the formal concept of capabilities in Belief-Desire-Intention agents and describe how agents can reason about these capabilities in order to reconfigure their plan library while retaining overall correctness constraints. To validate our approach, we show the implementation of our framework and an experiment using a practical example in the Mars rover scenario.

Keywords: Belief-Desire-Intention · modular architectures · autonomous agents · reconfigurability · robotic systems.

1 Introduction

Robots have been frequently used in real world applications over the years, from industrial robotics [30] to teleoperated robots in search and rescue [24]. However, there are still many open challenges such as: the German strategic initiative Industrie 4.0 that encourages research in the intelligent networking of machines; and robot assisted disaster response in the TRADR project [19]. The reconfigurability problem originally stemmed from manufacturing systems [18], but has since been expanded to self-reconfigurable robots [5,31] that can adapt to different situations via proper selection and reconfiguration of the functional components and the software that are available.

Due to the complexity present in these challenges modular architectures are typically employed to speed up and make the development of robotic systems easier. The Robot Operating System (ROS) [26] is an example of a popular middleware that can be used to develop a modular robotic system. In ROS, nodes are used to effectively capture robotic software in terms of a graph that describes the communication between distinct nodes. Some of the advantages of decoupling the system in this way include: more precise failure handling and

* Work supported by UK Research and Innovation Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE), and EP/R026084 (RAIN).

recovery mechanisms, since failures can be traced to individual nodes; and the complexity of the code is reduced when compared to monolithic systems, making it easier to add, replace, or remove functionality (i.e., nodes).

Agent-based control allows a system to dynamically adapt to changes in the environment through the use of modularity, decentralisation, autonomy, scalability, and reusability [20]. Many of these systems use cognitive agents, particularly those in the Belief-Desire-Intention (BDI) paradigm. Kohn and Nerode's MAHCA system [17] uses multiple knowledge-based agents as planners which generate the actions performed by the underlying control system. While these agents are not based on the BDI paradigm, which was only in its infancy when MAHCA was originally developed, the approach was designed to represent logical decision-making in a high-level declarative fashion. Recent agent-based approaches have been use in networks of autonomous agents interacting to solve complex and dynamic problems in manufacturing and supply chain decision making [22], and explored in the control of spacecraft [25], Unmanned Aircraft [33], and robotics [34]. Many of these approaches are explicitly BDI-based that aim to separate the symbolic and non-symbolic reasoning, and to model the mission designer's *intent*.

Modular architectures for robotic systems often require a framework for reconfigurability, allowing modules to be added and replaced should the need arise (e.g., in case of node failure). Several types of reconfigurability can be identified in these systems [8]: (i) reconfiguration at the hardware level, for instance the dynamic reconfiguration of effectors and sensors to cope with a hardware change; (ii) reconfiguration due to low-level control, such as ROS node reconfiguration and being able to replace nodes while still maintaining a working graph; (iii) reconfiguration due to high-level control, for example reconfiguring an agent's goals, plans, and knowledge.

In this paper, we introduce the reconfigurability of plan libraries in BDI agents [27]. Specifically, our approach is aimed towards modular architectures and applications that detect anomalies or malfunctions in a capability (an extended action specification) and can then reason about replacing it with alternative capabilities that are able to achieve the desired outcome. We use the BDI-based agent-oriented programming language Gwendolen [6] to implement our framework. By using this language, we can formally verify our new plan library using a program model checker, an important, and necessary [10], step towards the validation and reliability of the framework and its applications.

In the next section we provide a background on BDI agents and clarify the distinction between actions and capabilities. Section 3 introduces our reconfigurability framework, with an overview of the overall system and a running example. Then, we provide a formal description of capabilities and plan replacement in propositional logic. In Section 4 we describe our implementation of reconfigurability in the Gwendolen language, and then demonstrate its use in a practical experiment in the Mars rover scenario. Section 5 covers the related work, from purely theoretical approaches to application-based solutions. We end the paper in Section 6 with our conclusions and future work.

2 BDI Agent Programming Languages and Capabilities

Agents programmed using BDI languages commonly contain a set of *beliefs*, representing the agent's current state and knowledge about the environment in which it is situated; a set of *goals*, tasks that the agent aims to achieve; a set of *intentions*, tasks that the agent is committed to achieve; and a set of *plans*, courses of actions that are triggered by events. *Events* can be related to changes in the agents belief base or to the addition or removal of goals. The agent reacts to these events by creating new intentions which are generated from the set of applicable plans related to that event.

The body of a plan often contains a set of *actions* that can cause changes in the environment. Languages such as *Jason*, GWENDOLEN, and 2APL delegate these actions' specifications to the environment. Any preconditions or postconditions (i.e., effects) that such actions may have are invisible to the agent and are dealt with at the environment level. The environment may return some value to be unified with an open variable or, for example, in *Jason*it may also return a failure condition which will trigger the removal of the event that started the plan (plan failure).

It is possible to specify an action's pre and postconditions at the agent level using these languages, for example by creating a dummy plan with only one action, and writing the preconditions of the actions as the preconditions of the plan. The postconditions, if any, could then be written as belief operations (add or remove) to be executed after the action finishes. However, this is not how BDI agents are traditionally programmed in those languages and it can be difficult to reason about an action's pre and postconditions programmed this way.

These actions can also be modelled as *capabilities*. The main difference between them is the explicit specification of *pre* and *post*conditions. A capability can only be executed when its preconditions are true, and any postconditions that it has are added when the capability ends with success. This kind of action theory is also used in automated planning, known as actions in classical planning [12] or primitive tasks (also operators) in Hierarchical Task Network (HTN) planning [32]. BDI languages that implement this concept of capability include GOAL, 3APL, and 2APL (only for belief update actions).

The problem that we are interested in solving is improving the system's ability to adapt its behaviour in the event of a (software) failure or damage to some of its (physical) sub-systems in such a way that it can continue to achieve some, or all, of its goals. A BDI-agent is involved at the highest decision-making level of the system, and as such, we want the agent to be able to *recognise* that a component is no longer behaving as expected, then *invoke* diagnosis subsystems to identify the actions related to the failure, and lastly *reconfigure* the agent's plan library to cope with the failure. We note that failure detection and diagnosis in agents is touched on, for example, with semantics for adding duration and failure information in actions for the life-cycle of goals in [7], and through the use of trace expressions to specify protocols on top of sets of events (such as messages, beliefs, and actions) to be checked at runtime through the use of automatically generated monitors [11].

3 Reconfigurability Framework

Our framework is aimed towards systems that have a similar architecture to the one represented in Figure 1. The system goes through phases of potentially time critical operations followed by an offline phase in which it can reason about failure, perform reconfiguration, and if necessary reverify any relevant system properties. When such a system is deployed, the execution of its capabilities is monitored (e.g., with runtime monitors [11]). If a capability is detected as faulty, then the reconfiguration process tries to replace calls to the faulty capability with viable alternatives, which may include one or multiple capabilities.

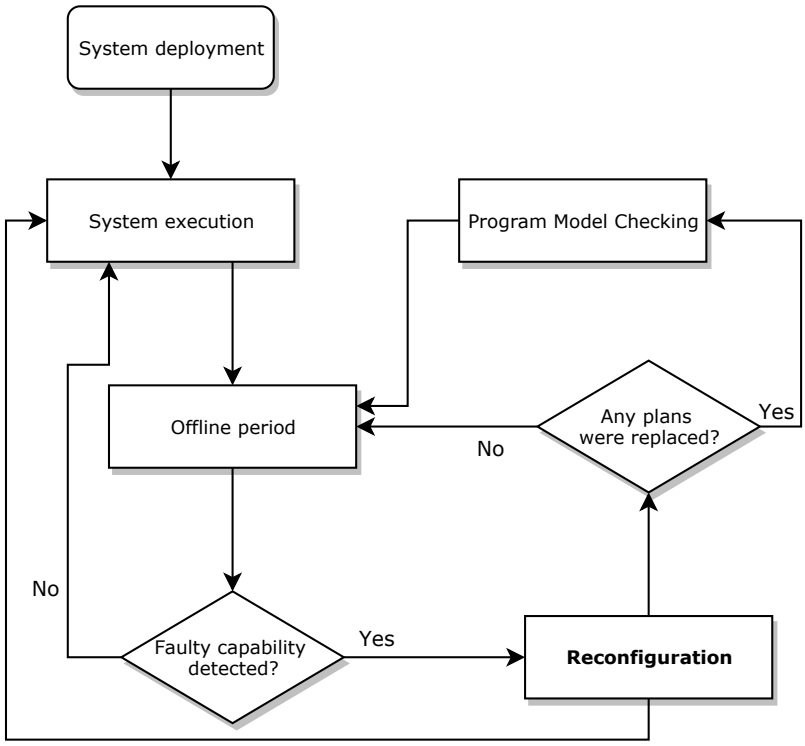

Fig. 1. Overview of the system.

We assume that the system has an offline period, which is very common in robotic systems with long-term autonomy (see [15] for example). During this period the robot can recharge and perform cleanup operations, among other things. But, more importantly from our point of view, it can also reconfigure itself if any faults were detected and verify any plans that were replaced (i.e., reconfigured) using, for example, a program model checker for agent programming languages.

We focus on the formal definition and implementation of the reconfigurability of plan libraries in BDI agent(s) that perform high-level reasoning within the system.

3.1 Running Example

Robots are increasingly deployed to explore hazardous environments that are dangerous for human exploration and often safety-critical, such as areas with extreme temperatures (monitoring of offshore structures [29]), lack of oxygen (both orbital [13] and planetary [35] space exploration), or high radiation (nuclear inspection and decommissioning [3]). Autonomous robots are especially important in scenarios with communication bottlenecks, for example, in planetary space exploration it can take a very long time for human operators to send commands from Earth to the robots. One such example is the Mars rovers used by NASA[1] in several missions. In this scenario an autonomous rover vehicle traverses the surface of Mars collecting image, soil, and rock data. For our example we assume that the rover has access to a topological map, which indicates areas of interest (denoted by waypoints) where the rover can collect data.

The Mars rover scenario can easily be seen as a system that matches the overview depicted in Figure 1. While in execution mode, the rover traverses through routes in the topological map, collecting data at each waypoint. The rover's offline period happens, for instance, during the night when it can not recharge using its solar panels and must conserve energy by remaining stationary.

Specifically, in this scenario we are interested in modelling the agent that is responsible for autonomously controlling the rover at a high-level. The actions that the agent can perform include moving between waypoints in a topological map, charging its batteries, collecting soil, collecting data, and taking images.

3.2 Preliminaries

Propositional Logic. We adopt a language of propositional logic $\mathcal{L}$ with formula ϕ defined over a finite set of literals, L, and with $\top, \bot \in \mathcal{L}$ denoting the *true* and *false* connectives respectively. We also use a finite set of abstract states S with element s and an entailment relation $s \models_{\mathcal{L}} \phi$ which defines when formula ϕ holds in s.

As previously mentioned in Section 2, actions form the body of a BDI plan. When these actions are executed, some parameters may need to be instantiated (this is typically done by unification). We use the notation $t\theta$ to indicate the application of a unifier, θ, to a term t.

3.3 Capabilities

Our formal representation of capabilities is based on the action theory found in classical automated planning, such as STRIPS reasoning [12], situation calculus [28], and the Planning Domain Definition Language (PDDL) [23]. As such,

[1] `https://mars.nasa.gov/`

our formalism is deliberately close to those, but a key difference is that we do not plan from scratch. We discuss the relationship between our work and planning systems in related work (Section 5).

A capability specification describes an action that an agent can take and any relation it has to the internal (self) and external (environment) facts of the agent. The specification is in the form of a set of preconditions and postconditions for the action. If the preconditions hold before the action is performed then the action specification states that eventually, if the action terminates with success, the postconditions will hold.

Capability Specification. We use the notation $\{C_{pre}\}\, C\, \{C_{post}\}$ where C is the capability, C_{pre} are the preconditions and C_{post} are the postconditions. C, C_{pre}, and C_{post} are all formulas $\phi \in \mathcal{L}$.

Capability Example. The *move* action of a rover can be represented as:

$$C = \{at(X),\, not\, X = Y\}\, move(X, Y)\, \{not\, at(X),\, at(Y)\}$$

such that X is the current position of the rover, and Y is the desired destination. Following the capability specification, we know that the rover must be $at(X)$ (precondition), and after the end of the execution it must be $at(Y)$ (postcondition).

Reasoning about the Execution of a Capability. The notation $\mathbf{do}(C\theta)$ indicates the execution of a capability, with its parameters instantiated by the θ unifier. The execution defines a transition on states in S that is completely specified by the specification of C. That is, if $s \in S$ and $s \models_{\mathcal{L}} C_{pre}\theta$ there is some unique state $s' \in S$ such that:

$$s \xrightarrow{\mathbf{do}(C\theta)} s' \text{ and } s' \models_{\mathcal{L}} C_{post}\theta$$

As discussed in Section 2, many BDI systems employ a simplistic action theory where the pre and postconditions of actions are not represented at the execution level of the agent. Thus, it is always possible to execute actions in those systems as long as the preconditions of the plan hold when the plan was selected. Our theory of plan validity (see next section) assumes that capability specifications are complete and correct and that the preconditions of a capability always hold when the system attempts to execute the associated action.

Execution Example. Suppose that in state s the proposition $empty(true)$ holds and the next action in the plan to be executed is the capability *collect_sample* with postcondition $empty(false)$:

$$s \xrightarrow{\mathbf{do}(collect\_sample)} s'$$

results in a state where $empty(false)$ holds.

3.4 Plans

A BDI plan is a structure which contains a sequence of capabilities as its body, but may also contain additional elements such as trigger events or guards. For simplicity, we ignore these additional elements in a plan body when reasoning about plan replacements, but we point out that most of these elements could be represented as capabilities.

Plan Specification. Given a plan, P, we write its preconditions as P_{pre} and its postconditions as P_{post}. We use the notation $\overline{C} = [C^1; C^2; \ldots; C^n]$ to indicate a sequence of capabilities that are to be executed as part of the body, $\overline{C}$, of a BDI plan. Our theory assumes that capabilities are guaranteed to execute sequentially, i.e., C^{i+1} is not executed until C^i_{post} holds.

Plan Body Example. We have a plan, P^1 to collect a rock sample at a particular position and then transmit the data. The body of this plan, $\overline{C^1}$, consists of a sequence of four capabilities:

$$\overline{C^1} = [move(X_1, Y_1);\ collect\_sample(S);\ move(Y_1, Y_2);\ transmit\_data(S)]$$

The first moves the rover to a position where it is capable of collecting a rock sample, performs the collection, then moves to a position where it can transmit the data, and finally performs the transmission.

Many BDI-based languages already allow the specification of preconditions in plans (e.g., plan context in Jason, or plan guards in GWENDOLEN), but it is unusual for a BDI plan to have explicit postconditions. However, we believe these can often be understood implicitly from the postconditions of the capabilities in a plan's body.

Pre and postcondition Example. Using the plan from the previous example, P^1, we can complete the plan specification by adding $P^1_{pre} = [at(X_1),\ empty(true)]$ and $P^1_{post} = [data\_transmitted(S)]$ as such:

$$P^1 = \{at(X_1),\ empty(true)\}\ \overline{C^1}\ \{data\_transmitted(S)\}$$

meaning that for the plan to be applicable the rover must be at waypoint X_1 and it must not be carrying any sample, and after the plan's conclusion the data of S will have been transmitted.

We represent pre and postconditions for plans explicitly because even though two capabilities may have different postconditions it may be the case that one can be replaced by another in a plan, without changing what the plan is intended to achieve. Since BDI plans are generally constructed by humans rather than automated planning systems we can not assume that the capabilities in the body of a plan are minimal with regards to the postconditions.

It may occur that the postcondition of a plan does not include all of the effects from the execution of the capabilities in the plan body (i.e., $P_{post} \neq C^1_{post} \cup \ldots \cup C^n_{post}$). For example, the proposition $at(Y_2)$ is not a postcondition

of P^1 since in this case we do not care if the final position of the rover is Y_2, only that it has successfully transmitted the data. This may be achievable from other waypoints, allowing the plan to be modified in a fashion that would have the rover transmitting data from a different waypoint.

Simple Plan Trace. A simple plan trace is one in which only the capabilities in the plan body cause state transitions in S. That is, the environment does not change apart from the execution of those capabilities and the plan's execution has not been interleaved with the execution of any other plan. Formally, let $[C^1; \ldots; C^n]$ form the body of some plan, P. Then a sequence of states $s_1; \ldots; s_{n+1}$ together with a unifier, θ, forms a simple plan trace for P if for all s_i, $s_i \xrightarrow{\mathbf{do}(C^i\theta)} s_{i+1}$.

3.5 Plan Replacement

When reasoning about plan replacement we assume an idealised execution environment for the plan represented by a simple plan trace. We ignore any impact that the environment (or another external factor) might have in the outcome of a capability, as these would be impossible to predict. The end system may still have them and they may make a plan that was replaced fail, but we do not address this issue in this paper, since this is not directly related to the reconfigurability problem.

Definition 1 (Valid Plan). *We say a plan P with a body consisting of the sequence of capabilities $[C^1; \ldots; C^n]$ is* valid *with regards to the specifications of the capabilities, if for all simple plan traces, $\langle s_1; \ldots; s_{n+1}, \theta\rangle$ where θ instantiates all the parameters of all the specifications for capabilities in the body of P, $s_i \models_{\mathcal{L}} C^i_{pre}\theta$ for all C^i. That is, the precondition in the specification for the next capability in the plan holds when some capability is executed.*

Note that plans in an actual BDI program may not be valid w.r.t. to the actions' specifications since they have been supplied by a programmer, not constructed from the specifications.

Definition 2 (Valid Plan Specification). *We say that a plan specification $P = \{P_{pre}\}\overline{C}\{P_{post}\}$ for a plan with body $[C^1; \ldots; C^n]$ is* valid *if*

1. *P is valid;*
2. *$P_{pre} \rightarrow C^1_{pre}$; and*
3. *for all simple plan traces, $\langle s_1; \ldots; s_{n+1}, \theta\rangle$ where θ instantiates all the parameters in the specifications of capabilities appearing in the body of P and all free variables in P_{pre} and P_{post}, if $s_1 \models_{\mathcal{L}} P_{pre}\theta$ then $s_{n+1} \models_{\mathcal{L}} P_{post}\theta$.*

The first point is covered in Definition 1. The second point states that if the preconditions of the plan holds *before* its execution, then the preconditions of the first capability in that plan's body also hold. The third point declares that a plan specification is valid if it can establish its postconditions on simple plan

traces. That is, if the precondition of a plan hold before execution of the plan, then the plan's postconditions hold *after* its execution.

Once again, we assume a static environment where all capabilities behave according to their specification. Note that this does *not* guarantee that the plan always works, only that it has been specified appropriately and sensibly programmed to work in most situations where it is invoked.

Definition 3 (Preservation of Plan Spec. Validity). *Consider a plan specification* $P = \{P_{pre}\}\overline{C}\{P_{post}\}$. *Let* P' *be a plan that is identical to* P *except that* P' *has body* $\overline{C'}$. *We say that* P' preserves the validity *of* P *if* $P' = \{P_{pre}\}\overline{C'}\{P_{post}\}$ *is valid.*

We argue that the preservation of plan specification validity is a minimal requirement when replacing plans. It states that if there is a static environment and no interleaved execution of plans, then the new plan will achieve the replaced plan's postconditions.

Definition 4 (Rational Plan Body Replacement). *We say the replacement of plan body* $\overline{C}$ *for* $\overline{C'}$ *in a plan* P *so it becomes a plan* P' *is* rational *if* P' *preserves the validity of* P.

Therefore we seek to implement mechanisms for plan body replacement that are rational.

Plan Body Replacement Example. If we detect a capability in the previous plan P^1 to be faulty, for example, the rover can no longer move from the place it collected the rock sample to a place to transmit the data due to battery deterioration. Then, the capability $move(Y_1, Y_2)$ can be exchanged in the plan body replacement:

$$\overline{C^{1'}} = \begin{bmatrix} move(X_1, Y_1);\ collect\_sample(S);\ move(Y_1, ChargeWaypoint); \\ recharge;\ move(ChargeWaypoint, Y_2);\ transmit\_data(S) \end{bmatrix}$$

with $ChargeWaypoint$ representing the waypoint where the rover can recharge its battery. This is a rational plan body replacement: the new plan $P^{1'} = \{at(X_1),\ empty(true)\}\,\overline{C^{1'}}\,\{data\_transmitted(S)\}$ is a valid plan which achieves P^1's postconditions whenever P^1's preconditions are true.

4 Implementation

We have implemented our theory as an extension of the Gwendolen programming language [6], chosen for its association with the Agent Java Pathfinder (AJPF) model-checker [9]. This provides a potential route for verification of a reconfigured plan library. To simplify implementation, we only use grounded capabilities in our practical experiment.

Algorithm 1 shows a high-level abstraction of our implementation for reconfigurability of plan libraries in BDI agents. Due to the availability and accessibility of implementations of fast classical planners, we opted to translate the search

for the replacement of plan bodies into a limited planning problem. Limited here refers to the use of a very small subset of information, instead of planning from scracth.

Algorithm 1: Implementation of plan library reconfigurability.

```
1  Function replace (capability)
2      Capabilities ← get_capabilities;
3      Capabilities ← Capabilities \ {capability};
4      if Capabilities = ∅ then
5          return false;
6      domain ← create_domain (Capabilities);
7      PlanLibrary ← get_plan_library;
8      Plans ← get_plans (capability);
9      while there exists {plan} ∈ Plans do
10         InitState ← propagate (plan, capability);
11         Goals ← get_post_cond (plan);
12         problem ← create_problem (InitState, Goals);
13         replacement ← STRIPS_planner (domain, problem);
14         if replacement = ∅ then
15             return false;
16         newplan ← replace_cap (plan, replacement);
17         PlanLibrary ← PlanLibrary \ {plan} ∪ {newplan};
18         Plans ← Plans \ {plan};
19     update_plan_library (PlanLibrary);
20     return true;
```

We start the reconfiguration of the plan library when a capability is detected to be faulty and in need of a replacement. First, we retrieve all capabilities that the agent has, except for the one that it wants to replace (lines 2–3). If we are left with an empty set of capabilities, then that capability cannot be replaced. Otherwise, the domain is created, translating all capabilities into STRIPS operators. Next, we fetch all plans from the agent's plan library that have the faulty capability in a plan's body (line 8).

In lines 9–18 we cycle through each of the plans that include the capability to be replaced. We construct the initial state from the propagation of the literals from the preconditions and postconditions starting at the first capability and going up to the last capability before the faulty one in the plan. This propagation is also known as progression in search algorithms. Our *Goals* set contains the postconditions of the plan to be replaced. We create the problem specification and then call a STRIPS planner to find the replacements by providing the domain and problem specifications that were translated from the Gwendolen syntax. Although any STRIPS planner would suffice, we chose the SIW+-then-BFSf

planner [21], one of the faster and top performing planners from the agile track in the International Planning Competition.

If no replacement is found by the planner, then the faulty capability cannot be replaced in that plan. Otherwise, we swap it with the replacement that was found (which can contain one or more alternative capabilities), remove the old plan from the temporary plan library, and add in the new plan. After we cycled all plans and replaced the faulty capability within them, the plan library is updated with the new modifications.

A plan in GWENDOLEN is started by an *event*, for example, a plan for completing a mission $mission1$ is activated when the goal (!) $mission1$ is added (+); this is known as a goal addition event. The plan will be selected and added to the agent's intention base if the formulae present in the *guard* (i.e., the context or precondition of the plan, goes after a colon and between curly brackets) are true. After a plan is selected, a sequence of actions in the plan body (denoted by ←) is executed.

4.1 Practical Experiment

We use a simple problem in our running example of the Mars rover scenario as a practical experiment. The problem is to replace a faulty movement capability, $moveW1W2$ that represents the route between the topological nodes $W1$ and $W2$. For this experiment we focus only on movement capabilities. Although other actions such as collecting rock data are also represented as capabilities, we omit them since they are not relevant to this experiment.

```
1  :Capabilities:
2  { at(waypoint1) } moveW1W2 { −at(waypoint1), +at(waypoint2) }
3  { at(waypoint2) } moveW2W1 { −at(waypoint2), +at(waypoint1) }
4  { at(waypoint1) } moveW1W3 { −at(waypoint1), +at(waypoint3) }
5  { at(waypoint3) } moveW3W1 { −at(waypoint3), +at(waypoint1) }
6  { at(waypoint3) } moveW3W2 { −at(waypoint3), +at(waypoint2) }
7  { at(waypoint2) } moveW2W3 { −at(waypoint2), +at(waypoint3) }
8  { at(waypoint1) } moveW1W4 { −at(waypoint1), +at(waypoint4) }
9  { at(waypoint4) } moveW4W1 { −at(waypoint4), +at(waypoint1) }
10 { at(waypoint2) } moveW2W5 { −at(waypoint2), +at(waypoint5) }
11 { at(waypoint5) } moveW5W2 { −at(waypoint5), +at(waypoint2) }
12 { at(waypoint5) } moveW5W4 { −at(waypoint5), +at(waypoint4) }
13 { at(waypoint4) } moveW4W5 { −at(waypoint4), +at(waypoint5) }
```

Fig. 2. Capabilities of the rover agent in GWENDOLEN.

The capabilities in Figure 2 represent the topological map that the agent has access to. A precondition list precedes the capability, which is followed by a postcondition list. The topological map consists of the following navigation routes between each waypoint: $W1 \Leftrightarrow W2$, $W1 \Leftrightarrow W3$, $W3 \Leftrightarrow W2$, $W1 \Leftrightarrow W4$, $W2 \Leftrightarrow W5$, and $W5 \Leftrightarrow W4$.

The plans for the rover agent are listed in Figure 3. There are plans for three different missions, each applicable when the agent is at a different location. For

example, the guard of $mission1$ (line 3) states that the agent must have the belief $at(waypoint1)$ expressing that the rover must be currently located in waypoint1. The body of $mission1$ (lines 5–8) contains the capabilities that must be executed in sequential order to successfully achieve the mission's goal.

```
1  :Plans:
2  +!mission1 [perform] :
3       { B at (waypoint1) }
4  ←
5       moveW1W2,
6       collect_soil,
7       moveW2W5,
8       collect_rock;
9  +!mission2 [perform] :
10      { B at (waypoint4) }
11 ←
12      moveW4W1,
13      collect_rock,
14      moveW1W2,
15      take_image;
16 +!mission3 [perform] :
17      { B at (waypoint3) }
18 ←
19      moveW3W1,
20      moveW1W2,
21      collect_rock,
22      moveW2W5,
23      take_image;
```

Fig. 3. Plan library of the rover agent in Gwendolen.

Figure 4 shows a simple example in the Mars rover scenario using a representation of all the capabilities described in Figure 2. A lander spacecraft stays in its original position and acts as a charging station for the rover, which starts next to the lander at waypoint 1 ($W1$). At some point during the system's deployment, either while in execution or in the offline period, the capability $moveW1W2$ is detected to be faulty. This could have been caused because, for example, the route between waypoint 1 and waypoint 2 is no longer valid (e.g., there is an unavoidable obstacle), or the route is consuming too much battery (e.g., the terrain became difficult to traverse).

The solution found by the planner was to replace the faulty capability $moveW1W2$ for $moveW1W3$ and $moveW3W2$. Then, we replace all occurrences of that capability in all plans, effectively removing the route between waypoint 1 and waypoint 2, and replacing it with the route from waypoint 1 to waypoint 3, and then from waypoint 3 to waypoint 2. Although this solves the problem caused by the faulty capability in all three mission plans, it also introduces some backtracking in the plan for $mission3$. This illustrates the trade-off between speed and optimality.

Because we are using an agile planner the solution is not always guaranteed to be optimal. However, as we previously mentioned, the translation from

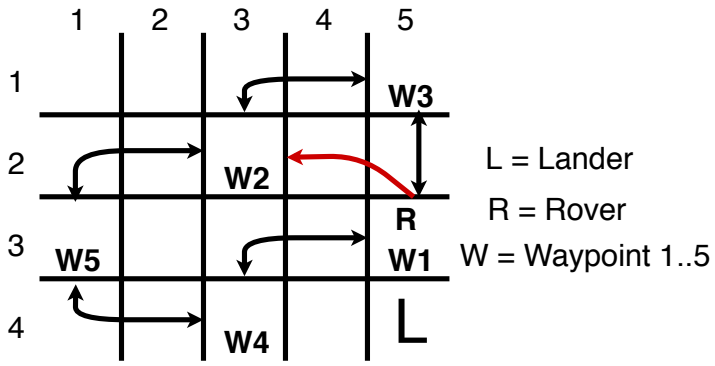

Fig. 4. Mars rover practical example.

Gwendolen can be used in any STRIPS classical planner, including optimal planners.

5 Related Work

In [16], an extension of a temporal epistemic logic is used to generalise model checking as a solution to reconfiguring reactive multi-agent systems. In this case the problem was to determine whether a set of reactive robots can combine into a robot that satisfies the functionality of the system. Two scenarios are given, one is a monolithic system and the other is an individual module that is part of a bigger system. They also defined a new logic-based language to represent multi-agent systems and reason about reconfigurability at an abstract level, but our approach is intended as a generic extension applicable to a range of BDI-based agent-oriented programming languages and which, as we have shown, can result directly in an implemented system.

An agent-based framework is proposed for resource reconfiguration in production lines of industrial assembly applications considering product specification and capabilities of production resources [2]. The reconfiguration is goal-based and done through task reallocation. The authors claim that the framework is implemented and runs on a real-world assembly system, however, there is no formal description of the framework or any of its features. Although the concept of reconfigurability and capability is similar to ours, the main difference is that their concepts are intrinsically tied to industrial assembly applications, whilst our framework is domain independent.

An architecture for planning in reconfigurable manufacturing systems is presented in [4]. Reconfiguration in these systems are described to occur in three different scenarios: a production change, physical malfunctions, or a change in production goals. The control system implements a sense-plan-act cycle using ontology-based knowledge to regenerate the planning domain specification when

necessary. Similar to the previous approach, this architecture is application specific, and thus, it does not address generic reconfigurability problems.

A reconfigurable agent-based architecture for use in autonomous nuclear waste management is reported in [1]. In this system a BDI-agent controls a ROS-based system for sorting and segregating different types of low radiation level nuclear waste. Reconfiguration is handled by pre-existing plans in the BDI agent rather than by the agent reconfiguring its existing plans. This necessarily limited the extent to which the system could adapt to hardware degradation and changes in its environment.

The reconfigurability scenarios that we described could be represented as replanning problems or plan repair problems [14]. We are particularly interested in applications where a high degree of assurance (ideally formal verification) is required. Formal verification of planning is still an area in its infancy while the verification of BDI agents is well studied — hence we have developed a framework in which planning is used to instantiate new replacement plans[2] in BDI programming languages. Furthermore the complete state of the world would have to be passed to a planner. By using our reconfigurability framework this can be avoided, potentially saving computation time.

6 Conclusions

There are different ways that the reconfigurability problem can be solved, such as: preemptively adding plans that cover plan failure; or replanning from scratch. However, the former is prone to human error, and the latter can take substantially longer in complex problems.

In this paper, we have described a formal framework for plan library reconfigurability in BDI agents. We presented a theory based on capabilities and plans, and introduced several definitions concerning how to reason about valid plan replacement. Further to this, we implemented our framework into the Gwendolen BDI language and used an agile planner to find capability replacements that are then merged into a plan replacement. As a demonstration of the implementation of our framework we performed a practical experiment on reconfigurability in the Mars rover scenario.

The performance of the implementation our reconfigurability framework is intrinsically tied to the performance of the planner's implementation that we used to find the proper replacements for a faulty capability. Therefore, future experiments to measure the scalability of our framework should include different planners to better evaluate how well our approach scales by isolating the performance of the planning component. Future work also include considering plan regression to rationally discard redundant capabilities that came before the faulty capability to remove any unnecessary backtracking.

[2] The areas of automated planning and BDI agent programming both use the word "plan" but with slightly different meanings.

References

1. Aitken, J.M., Veres, S.M., Shaukat, A., Gao, Y., Cucco, E., Dennis, L.A., Fisher, M., Kuo, J.A., Robinson, T., Mort, P.E.: Autonomous nuclear waste management. IEEE Intelligent Systems **33**(6), 47–55 (Nov 2018)
2. 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)
3. Bogue, R.: Robots in the nuclear industry: a review of technologies and applications. Industrial Robot: An International Journal **38**(2), 113–118 (2011)
4. 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)
5. 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)
6. Dennis, L.A., Farwer, B.: Gwendolen: A BDI language for verifiable agents. In: Logic and the Simulation of Interaction and Reasoning. AISB, Aberdeen (2008)
7. Dennis, L.A., Fisher, M.: Actions with durations and failures in BDI languages. In: 21st European Conference on Artificial Intelligence. vol. 263, pp. 995–996. IOS Press (2014)
8. Dennis, L.A., Fisher, M., Aitken, J.M., Veres, S.M., Gao, Y., Shaukat, A., Burroughes, G.: Reconfigurable autonomy. KI - Künstliche Intelligenz **28**(3), 199–207 (Aug 2014)
9. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated Software Engineering **19**(1), 5–63 (2012)
10. Farrell, M., Luckcuck, M., Fisher, M.: Robotics and integrated formal methods: Necessity meets opportunity. In: International Conference on Integrated Formal Methods. pp. 161–171. Springer (2018)
11. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Verifying and validating autonomous systems: Towards an integrated approach. In: Proceedings of the 18th International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 11237, pp. 263–281. Springer (2018)
12. 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)
13. Flores-Abad, A., Ma, O., Pham, K., Ulrich, S.: A review of space robotics technologies for on-orbit servicing. Progress in Aerospace Sciences **68**, 1–26 (2014)
14. Fox, M., Gerevini, A., Long, D., Serina, I.: Plan stability: Replanning versus plan repair. In: Proceedings of the 16th International Conference on Automated Planning and Scheduling. pp. 212–221. AAAI Press, Cumbria, UK (2006)
15. Hawes, N., Burbridge, C., Jovan, F., Kunze, L., Lacerda, B., Mudrova, L., Young, J., Wyatt, J., Hebesberger, D., Kortner, T., Ambrus, R., Bore, N., Folkesson, J., Jensfelt, P., Beyer, L., Hermans, A., Leibe, B., Aldoma, A., Faulhammer, T., Zillich, M., Vincze, M., Chinellato, E., Al-Omari, M., Duckworth, P., Gatsoulis, Y., Hogg, D.C., Cohn, A.G., Dondrup, C., Fentanes, J.P., Krajnik, T., Santos, J.M., Duckett, T., Hanheide, M.: The STRANDS project: Long-term autonomy in everyday environments. Robotics Automation Magazine **24**(3), 146–156 (2017)
16. Huang, X., Chen, Q., Meng, J., Su, K.: Reconfigurability in reactive multiagent systems. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence. pp. 315–321. AAAI Press, New York, USA (2016)

17. Kohn, W., Nerode, A.: Multiple agent autonomous hybrid control systems. In: Proc. 31st Conf. Decision and Control (CDC). pp. 2956–2964. Tucson, USA (1992)
18. Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G., Brussel, H.V.: Reconfigurable manufacturing systems. CIRP Annals **48**(2), 527–540 (1999)
19. Kruijff-Korbayová, I., Colas, F., Gianni, M., Pirri, F., de Greeff, J., Hindriks, K., Neerincx, M., Ögren, P., Svoboda, T., Worst, R.: TRADR Project: Long-term human-robot teaming for robot assisted disaster response. KI - Künstliche Intelligenz **29**(2), 193–201 (Jun 2015)
20. Leitão, P.: Agent-based distributed manufacturing control: A state-of-the-art survey. Engineering Applications of Artificial Intelligence **22**(7), 979–991 (2009)
21. Lipovetzky, N., Ramirez, M., Muise, C., Geffner, H.: Width and inference based planners: SIW, BFS (f), and PROBE. In: Proceedings of the 8th International Planning Competition (2014)
22. Marik, V., McFarlane, D.: Industrial adoption of agent-based technologies. IEEE Intelligent Systems **20**(1), 27–35 (Jan 2005)
23. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL - the planning domain definition language. Tech. Rep. TR-98-003, Yale Center for Computational Vision and Control (1998)
24. Murphy, R.R.: Trial by fire [rescue robots]. IEEE Robotics Automation Magazine **11**(3), 50–61 (Sept 2004)
25. Muscettola, N., Nayak, P.P., Pell, B., Williams, B.: Remote agent: To boldly go where no AI system has gone before. Artificial Intelligence **103**(1-2), 5–48 (1998)
26. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: Workshop on Open Source Software at the International Conference on Robotics and Automation. IEEE, Japan (2009)
27. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proceedings of the first International Conference on Multi-Agent Systems. pp. 312–319 (1995)
28. Reiter, R.: The frame problem in situation the calculus: A simple solution (sometimes) and a completeness result for goal regression. In: Lifschitz, V. (ed.) Artificial Intelligence and Mathematical Theory of Computation, pp. 359–380. Academic Press Professional, Inc., San Diego, CA, USA (1991)
29. Shukla, A., Karki, H.: Application of robotics in offshore oil and gas industry – A review Part II. Robotics and Autonomous Systems **75**, 508–524 (2016)
30. Singh, B., Sellappan, N., P., K.: Evolution of industrial robots and their applications. International Journal of Emerging Technology and Advanced Engineering **3**(5), 763–768 (May 2013)
31. Støy, K., Brandt, D., Christensen, D.J.: Self-Reconfigurable Robots. MIT Press (2010)
32. Tate, A.: Generating project networks. In: Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2. pp. 888–893. IJCAI'77, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1977)
33. Webster, M.P., Fisher, M., Cameron, N., Jump, M.: Formal methods for the certification of autonomous unmanned aircraft systems. In: Proc. 30th Int. Conf. Computer Safety, Reliability and Security. LNCS, vol. 6894, pp. 228–242. Springer (2011)
34. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Programming Multi-Agent Systems, LNCS, vol. 7837, pp. 54–71. Springer (2013)
35. Wilcox, B.H.: Robotic vehicles for planetary exploration. Applied Intelligence **2**(2), 181–193 (1992)