Improving plan execution robustness through capability aware maintenance of plans by BDI agents

Alan White*, Austin Tate and
Michael Rovatsos

Artificial Intelligence Applications Institute,
Centre for Intelligent Systems and their Applications,
School of Informatics,
University of Edinburgh, UK
Email: a.g.white@sms.ed.ac.uk
Email: a.tate@ed.ac.uk
Email: mrovatso@inf.ed.ac.uk
*Corresponding author

Abstract: In a realistic environment, intentions of belief-desire-intention (BDI) agents may be threatened by exogenous change. Subsequent activity failure may incur debilitative consequences that hinder both recovery and subsequent goal achievement. Capability aware, maintaining plans (CAMP-BDI) embodies BDI agents with capability knowledge, allowing anticipation of threats to activity success and stimulating the proactive, preventative modification of intended plans. We describe resultant agent-level algorithms and supporting architecture, including extension to provide decentralised, distributed maintenance through structured messaging. Our results show superior goal achievement to a reactive equivalent in a stochastic environment, increasing with the likelihood of debilitative failure effects. We suggest CAMP-BDI offers a valuable approach towards robustness, particularly in tandem with reactive recovery methods.

Keywords: multi-agent teamwork; belief-desire-intention; BDI; planning, capability, robustness.

Reference to this paper should be made as follows: White, A., Tate, A. and Rovatsos, M. (2017) 'Improving plan execution robustness through capability aware maintenance of plans by BDI agents', *Int. J. Agent Oriented Software Engineering*, Vol. 5, No. 4, pp.306–335.

Biographical notes: Alan White received his Bachelor's degree in Computer Science from the University of Strathclyde in 2004, and PhD in Informatics from the University of Edinburgh in 2017. His main research interest lies in the application of multiagent systems within realistic environments, with a focus upon self-organisation and robustness.

Austin Tate is the Director of the Artificial Intelligence Applications Institute (AIAI) and holds the Personal Chair of Knowledge-Based Systems at the University of Edinburgh. He is a Fellow of the Royal Academy of Engineering, Fellow of the Royal Society of Edinburgh (Scotland's National Academy), Fellow of the Association for the Advancement of AI, Fellow of the British Computer Society, and on the editorial board of a number of AI journals. His internationally sponsored research work is focused on emergency response and involves advanced knowledge and planning technologies, and collaborative systems especially using virtual worlds.

Copyright © 2017 Inderscience Enterprises Ltd.

Michael Rovatsos is a Reader at the School of Informatics of the University of Edinburgh, where he has been leading the Agents Research Group since 2004. His research is in multi-agent systems, with a particular focus on algorithms for reasoning about interaction among intelligent agents. He has published over 85 papers in this area, and his work has attracted more than €10m of external funding to date. He has been involved in the organisation of over 80 international scientific events, and regularly serves on the committees of most major AI conferences.

This paper is a revised and expanded version of a paper entitled 'CAMP-BDI: A Pre-emptive Approach for Plan Execution Robustness in Multiagent Systems' presented at 18th International Conference on Principles and Practice of Multi-Agent Systems, Bertinoro, Italy, 26–30 October 2015.

1 Introduction

The belief-desire-intention (BDI) model has become a de-facto standard for development of intelligent agents, employed within realistic, stochastic and dynamic domains such as emergency response. Exogenous change may occur during plan execution in these types of environment, contradicting assumptions underlying initial plan formation and consequently increasing the risk of activity failure(s). Current BDI implementations often use reactive failure handling methods such as replanning, plan repair, or execution of predefined failure recovery plans – but failure may be associated with debilitative consequences that can stymie such recovery. Continuous planning can handle initial uncertainty by postponing planning decisions, but such shorter-term decision making risks inadvertent long-term failure – for example, failing to identify (reserve) key resources in advance, which are subsequently lost to contention before their necessity *is* identified.

The paper describes the capability aware, maintaining plans (*CAMP-BDI*) approach for performing proactive plan repair (or *maintenance*) in response to where exogenous change during execution threatens intended plans. CAMP-BDI agents form long term plans but are also embodied with capability meta-knowledge; allowing introspective reasoning to identify failure risks *and* the advance reservation of required resources. The following contributions are presented:

- an algorithm for anticipatory plan repair behaviour, referred to as performance of *maintenance*
- extension of local behaviour to encompass decentralised maintenance of distributed intentions
- a supporting architecture providing the capability, dependency, and obligation knowledge used to perform introspective reasoning and guide maintenance changes
- a policy mechanism allowing runtime tailoring of maintenance behaviour
- a policy mechanism allowing runtime tailoring of maintenance behaviour.

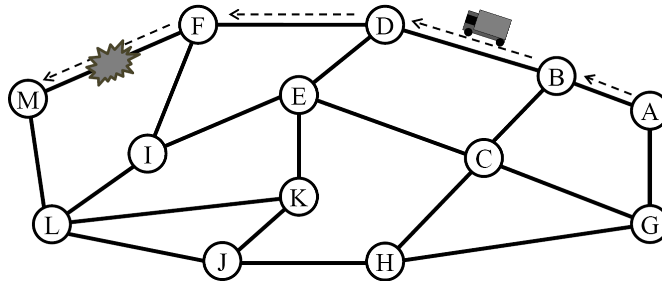
We experimentally evaluated CAMP-BDI within a logistics environment, similar to our motivating example, in comparison against reactive replanning. Results over multiple

experimental runs, for scaling failure-debilitation probabilities, showed CAMP-BDI offered superior goal achievement over replanning when failure risked debilitative consequences. CAMP-BDI also offered superior planning efficiency (less planner calls per goal achieved) as post-failure debilitation became increasingly probable (reflecting the increasing difficulty of reactive recovery).

2 Motivating example

Our motivating example describes a logistics domain; the multiagent system (MAS) attempts to transport cargo objects between locations, within a stochastic, dynamic, continuous and non-deterministic environment. Uncertainty arises from suboptimal agent health or exogenous change – including flooding, landslips, or emergence of 'danger zones' which prohibit activity in certain locations. Delivery goals are achieved through the formation and execution of distributed plans by heterogeneous agent teams; agent types include *Trucks* (cargo transporters), support vehicles (including *Bulldozers* that clear blocked roads or *APCs* – armoured personnel carriers – that remove danger zones), and logical agents acting as organisational controllers or brokers.

Figure 1 Example of truck executing a plan to travel from *A* to *M*



In our example (Figure 1), activity failure risks consequent debilitation – hindering both recovery and future goal achievement. Here, a *Truck*, travelling from *A* to *M*, sees its intended *move*(*F*, *M*) activity threatened by flooding on the road $F \rightarrow M$. Recovery after failing *move*(*F*, *M*) at *F* requires costly backtracking to use an alternate route; proactive behaviour may avoid this cost by identifying a need to change route earlier. Certain states may also increase the risk of failure without *ensuring* it – e.g., if $F \rightarrow M$ was *partially* flooded.

CAMP-BDI aims to improve robustness – expressed in terms of goal achievement – in realistic environments where exogenous change may occur, failure may risk debilitation, and domain complexity renders probabilistic methods intractable. We argue this requires proactive anticipation and avoidance of failure, through preventative plan modification, on both a single agent and multiagent team level. Operator models used for deterministic planning in such realistic environments will represent an approximation – our approach should recognise where states increase failure risk, but not at the significance required for inclusion within operator preconditions.

3 Architecture components

CAMP-BDI agents require meta-knowledge to introspectively reason over intended plans (i.e., our maintenance algorithms require agents to be 'capability aware'). We regard these knowledge components as a subset of *Beliefs*, although their semantics will be implementation specific. CAMP-BDI agents must also distinguish between a selected desire and the associated plan when considering intentions, as we wish to modify the latter to ensure achievement of the former. This leads us to adopt the definition expressed by Simari and Parsons (2006), viewing an intention as combining a goal *and* associated plan – i.e., $i = \{goal_i, plan_i\}$.

3.1 Capabilities

Agents hold *Capability* meta-knowledge defining which activities – and ergo goals – they can achieve, used within our algorithms to anticipate and avoid failure. A common model is used to represent both activities performed by the agent and those requiring delegation; this allows out CAMP-BDI agents to employ the same introspective reasoning logic for both locally performed and delegated activity (as both types may exist within the same $plan_i$).

We define an activity a as equivalent to a *task* within a hierarchical task network (HTN); i.e., a may be composite (a sub-goal which requires decomposition into a subplan) or primitive (an atomic action). An a is viewed as a deterministic state transition $F(a, S) = S'$; i.e., (successfully) executing a in state S achieves state S'. We refer to the agent's beliefs at the point of execution for a (the *execution context* of a) as B_a. A plan p for a goal g is an ordered sequence of n activities ($p\{a_1,\ldots a_n\}$) whose effects ultimately achieve g; if agents use continual planning, p may contain composite activities whose decomposition is deferred until execution.

A capability $c(a)$ held by an agent provides knowledge about that agent's ability to perform a, and contains the following fields:

$$c(a) = \langle s, g(a), pre(a), eff(a), conf(a, B_a) \rangle$$

- A signature s of name n with x parameters; $s = n(t_1,\ldots,t_x)$. Each capability instance in a MAS is uniquely identifiable through combining s with the holding agent's identifier.
- $g(a)$ defining the goal state (S_G) achieved by executing a – i.e., if a can be represented as $F(a, S) = S'$, then $S_G \subseteq S'$. Initially abstract, $g(a)$ is ground using the parameters (t) of s.
- The preconditions $pre(a)$ for performing a through this capability – i.e., a set of state atoms which must be true (i.e., $pre(a) \subseteq B_a$) to avoid *guaranteed* failure. Both this and $eff(a)$ are specified as abstract, but can be ground based upon the parameter (t) values of s.
- The complete set of post-effects of a, $eff(a)$, including side-effects (i.e., giving $eff(a) = g(a) \cup sideeffects(c(a))$); allowing determination of S' for $f(a, S)$. Different capabilities may achieve the same $g(a)$ with different side-effects – denoting the

semantic differences between performing a *move* activity through (for example) *flying* or *driving*.

- A *confidence* function used to estimate the quality (i.e., the likelihood of success) of performing a, using $c(a)$, given B_a. Returns a scalar value; $conf$: $a \times B_a \rightarrow [0: 1]$.

3.1.1 Capability typology

Capability type is defined through two overlapping categories; *granularity* – whether a (and $c(a)$) is a *primitive* or *composite* – and *locality* – whether the agent can perform a ($c(a)$ is *internal*) or knows of some agent it can delegate a to ($c(a)$ is *external*).

3.1.1.1 Primitive capabilities

Primitive capabilities indicate the holding agent can achieve $g(a)$ through a single, atomic a [similar to *know-how* of a *basic activity* as defined by Singh et al. (2010)]. In order to be executable, all plans must eventually resolve to some set of atomic activities – in distributed plans, this may require over multiple levels of decompositional delegation. Consequently, composite and external capability knowledge ultimately derives from some subset of primitive capabilities. Similarly to classical planning operators, primitive capability knowledge must be defined (by the agent programmer) at implementation time.

3.1.1.2 Composite capabilities

Composite capabilities represent agent awareness of *at least one* plan to perform a and achieve $g(a)$; i.e., a is non-atomic (divisible), representing a root $goal_i$ or sub-goal within a plan. This type enables reasoning over abstract (*composite*) activities (sub-goals) in plans, particularly if agents use continual planning (i.e., will refine certain abstract activities during plan execution).

BDI agents typically utilise libraries of pre-formed plans. Composite capabilities provide a 1: n mapping for $g(a)$ against associated library plans (i.e., represent knowledge of $n > 1$ plans for achieving $g(a)$). Conversely, each library plan maps to exactly one composite capability (i.e., one $g(a)$), with the capability field values being derived from the (n) plans being represented.

The preconditions ($pre(a)$) define all states under which a plan can be selected, and consequently are formed as the disjunction of all the represented plans' selection conditions (we assume if selection conditions for plan p hold, so do those for all $a \in p$). The $eff(a)$ field is set as the achieved goal states ($g(a)$); this is a generalisation, as the semantics of *how* $g(a)$ is met (which plan would be selected, with what side-effects) will vary with individual a instances and their execution context B_a. The exact $eff(a)$ can be formed for a given a and B_a through determining which plan would be selected for that *specific* context; if multiple plans are viable, we assume that with greatest estimated confidence would be selected.

3.1.1.3 External capabilities

CAMP-BDI agents are expected to advertise capabilities where they can accept obligations from others (with any authority constraints reflected by selective advertisement, and updated to reflect changes in circumstance such as confidence loss);

recipients use the received information to form a corresponding *external* capability set, representing where an a can be performed through delegation. Both primitive and composite capabilities can be advertised, although knowledge of the plans represented by the latter is restricted to the advertiser. As obligants identify and perform any plan selection within their internal reasoning, external capabilities are always regarded as *primitive* – i.e., will be indivisible from the (potential) dependant's perspective.

3.1.2 The confidence function

It is impossible to represent every state that may cause activity failure within deterministic preconditions, as this would constrain operators to be virtually unusable [McCarthy (1958) describe this as the 'specification problem']. In such a context, preconditions define states where *success* is not *guaranteed*, but is instead *probable*. Our maintenance process uses *capability confidence functions* to account for states that increase failure risk, but which are still not considered significant enough to represent within preconditions.

The confidence function ($conf_a$ (a, B_a)) provides a scalar estimate of quality – used to identify whether changes to B_a have increased failure risk, even if a's preconditions still hold. Use of a numerical value (0…1) allows comparison between capabilities sharing the same s without requiring awareness of semantic differences in their confidence estimation. Numerical estimation also enables flexible granularity – i.e., the function implementation can provide a Boolean (e.g., {0 = *false*, 1 = *true*}), enumerated (e.g., {*succeed* = 1, *maybe* = 0.5, *fail* = 0}) or specific probabilistic estimate, depending upon knowledge and computation constraints.

The semantics of $conf_a$(a, B_a) depend upon both the capability type and a itself. For an unground a, confidence estimation indicates *general* ability for that activity type – i.e., for achieving $g(a)$ in B_a. A ground a facilitates use of additional semantic information to provide a confidence value *specific* to that a. *Primitive* capabilities require predefined implementation for both estimation types – e.g., using historical records [Singh et al. (2010) use a similar methodology to learn plan reuse contexts] or programmer knowledge (requiring supporting domain analysis).

3.1.2.1 Primitive capability confidence estimation

Primitive capability confidence estimation depends upon both the capability holding (a performing) agent and the operating environment. Implementation may be aided through the same processes of state analysis as required to form deterministic planning operators – i.e., the identification of which states impact success and to what level of significance. We suggest this can reduce the specification burden, as such information will likely be necessary to specify plan executing agents (and plans) regardless of whether our approach is being used. Maintenance policies (described in 3.3) can also provide a means to compensate for any consistent over or under estimation by the confidence function.

3.1.2.2 External capability confidence estimation

Agents are unlikely to possess the semantic knowledge (or sensory awareness) to perform confidence estimation for *external capabilities*. Instead, such estimations are provided as fixed values by the (potential or actual) obligant performing a. These confidence values

indicate general confidence in the case of capability advertisements, but can provide specific estimates for agreed contracts through the external capability field (described in 3.2).

3.1.2.3 Composite capability confidence estimation

Composite capabilities represent knowledge of a set of plans P_{cc} for $g(a)$, such that estimation of composite capability confidence utilises the estimated confidence for each $p \in P_{cc}$ where p can be selected given B_a – i.e., the quality of plans the agent knows for performing a in B_a. Estimation of confidence in an *individual* plan is both relevant for threat anticipation (to estimate whether sub-goals or composite activities can be refined to an acceptable level of quality) and maintenance (to decide whether to accept maintenance plans generated to address an identified threat – see 4.3).

Confidence in a plan p derives from the activities $a \in p$, and may be calculated in various ways. For example, the minimum confidence for any $a \in p$ may provide the confidence for p (similar to TÆM's *q_min* metric), as described below. B_p is the execution context for the first activity – $a1$ – in p, and B_a the estimated execution context of a_n 2 p. After confidence is estimated for each a_n, its (capability-defined) effects are added to B_a, giving the estimated execution context for the following activity in p (i.e., for a_{n+1}):

$$conf_{\min}\left(p, B_p\right) = \min_{a \in p} conf\left(a, B_a\right)$$

The $conf_{\min}$ estimation can be employed where every $a \in p$ must have acceptable confidence. This may be considered over-constraining, as the confidence 'score' will be determined solely by the single worst activity rather than *how* many activities are of a low confidence. Such constraints may be desirable to guard against future maintenance requirements, by ensuring a plan will not be accepted if any constituent activity is of low (i.e., maintenance requiring) quality (although exogenous change may still necessitate further change). One alternative is to use *averaged* activity confidence ($conf_{\text{avg}}$):

$$conf_{avg}\left(p, B_p\right) = \left(\frac{\sum_{i=1}^{n} conf\left(a_i, B_{a_i}\right) \times w_i}{\sum_{i=1}^{n} w_i} \right)$$

The above includes a *weighting* function ($w_i \rightarrow \mathbb{Z}_{\geq 0}$) to scale the significance of activity a_i in contributing to confidence of p (where p is formed of n activities). For example, w_i may be employed to give greater significance to more immediate activities (e.g., $w_i = (n - 1)/i$), reflecting that the risk of exogenous change increases uncertainty when estimating the execution context – and confidence – of later activities. This may also be advantageous where it is difficult to generate acceptable plans under the constraints defined by $conf_{\min}$; using an averaged value can allow incremental improvement in $plan_i$ confidence, with any inserted low-confidence activities being individually addressed by maintenance in subsequent reasoning cycles.

Composite capability confidence derives from the plans represented by that capability. Composite estimation returns the confidence value estimated for the *highest confidence* plan (where selection preconditions hold), and zero if *no* plans could be

selected given the execution context. This is described below, where a_{goal} is the activity being performed through the composite capability and $B_{a_{goal}}$ the execution context:

$$conf\left(a_{goal}, B_{a_{goal}}\right) = \max_{pre(p) \subset B_{a_{goal}}} p \in P_{capability}\, conf\left(p, B_{a_{goal}}\right)$$

Composite capability estimation effectively sees formation of an AND-OR tree [similar to goal-plan trees in Thangarajah et al. (2003)], representing all potential plan and subplan paths to decompose and execute a_{goal}. The estimated value requires estimating the confidence of every leaf activity ($O(n)$ worst-case complexity, for n leaf nodes), where each leaf activity is primitive, and therefore originates from a primitive or external capability confidence value. We assume the decompositional nature of plans prevents cyclical loops – this property would be further required for agent activity itself (e.g., to avoid infinite looping).

Considerable scope for domain specific optimisation exists for both primitive and composite types of confidence estimation, depending upon agent and environment details. For $conf_{\min}$, $\alpha - \beta$ pruning may improve common case complexity where plan confidence values are being evaluated against a minimum value threshold (i.e., for policy maintenance). Composite capabilities may also represent runtime planning abilities, with associated confidence estimation requirements; one possibility is to employ methods similar to heuristic planning, such as *relaxing* a_{goal} to form a plan that can then provide the basis for approximate estimation.

3.2 Obligation and dependency contracts

We assume agents form dependency contracts in advance to counter potential contention over agent (i.e., to use advertised capabilities) and environmental resources. CAMP-BDI agents must be aware of their obligations to and dependencies upon others; contracts define the mutual beliefs established between agents in such delegation relationships, with definition of the following fields being required during contract formation:

a The *activity*, agreed by the obligant(s) to be performed upon request by the dependant. We use *dependant intention* to refer to the dependant's $plan_i$ containing that delegated activity.

b *Causal link* states that the dependant will establish (i.e., as effects of preceding activities in the dependant intention) prior to requesting execution of the delegated activity.

c An *external capability* defined by the obligant in order to detail anticipated post-effects and confidence for executing the delegated activity, using causal link states to estimate execution context. If multiple obligants are involved, their individual capabilities are merged to form the external capability, with:

- confidence as the minimum confidence held by an individual obligant
- preconditions as the conjunction of all individual preconditions
- effects as the union of all obligant post-effects.

d A *maintenance policy*, whose contents and usage are detailed below.

3.3 Maintenance policies

Policies allow dynamic regulation of system behaviour, without requiring modification of the underlying implementation (Tonti et al., 2003) – such as to define constraints or relaxations for activities or goals. In CAMP-BDI we use *maintenance policies*, applied to a stated set of agents and/or capabilities, to define key variables influencing our maintenance behaviour; specifically, a *threshold* stating the minimum acceptable confidence for an activity and a *Priority* used to define relative prioritisation where multiple activities in an $plan_i$ are (identified as) threatened.

Maintenance policy fields can be used to balance the costs of proactive maintenance; activities associated with more severe failure consequences can be given lower confidence thresholds, to increase the likelihood of CAMP-BDI maintenance attempting confidence raising plan modification. Conversely, activities with lesser failure consequence can be given lower thresholds or priorities – reducing the probability of proactive maintenance to instead rely upon reactive recovery. This can extend to disabling maintenance for activities with negligible consequences, or enabling maintenance during runtime if that assumption is found to be false.

Use of a policy mechanism also facilitates runtime modification of such values – unlike implementation time definition, maintenance behaviour can be adapted (by a human user or some automated process) as new environmental or agent performance behaviour is learned. This principle of externalising behavioural influences (through policies) also provides a framework that can aid the genericisation and reuse of CAMP-BDI agents, and suggests options for future research and development.

Contract maintenance policies must merge the – potentially divergent – maintenance policies applicable at both the obligant(s) and dependant side, in order to provide a policy specific to that activity delegation. The dependant policy will be that associated with the dependant $goal_i$; i.e., where the associated $plan_i$ contains the delegated activity. The obligant policy(s) will be those associated with that agent and the delegated activity – which will itself correspond to an $goal_i$ (and $plan_i$) adopted by that agent in order to perform that obligated activity.

The merge process selects the lowest threshold value and highest priority value from the dependant and obligant maintenance policies to form the equivalent fields in the merged policy – i.e., the merged policy contains the most constraining values upon maintenance. These shared values ensure obligants will have performed maintenance before updating dependants of any confidence changes, providing a means to ensure a minimal subset of the delegated plan will be modified; any conditions for triggering maintenance of the *dependant* $plan_i$ will always also apply to maintenance of associated obligant plans.

4 The CAMP-BDI algorithm

CAMP-BDI extends the generic BDI reasoning cycle defined by Rao and Georgeff (1995). Algorithm 1 shows our insertion of steps to perform dependency contract formation, and for performing maintenance of intended plans through the *maintain* function. Three contexts are defined for this invocation of *maintain*:

1 after the intention *i* is selected

2 upon receipt of an *obligationMaintained* messages from obligants (i.e., following the consequent updating of dependency knowledge)

3 when the agent has no selected intentions (i.e., is otherwise idle).

The third context uses maintenance to update mutual beliefs in delegated activities, through examining (and maximising) confidence in preformed or cached plans for obligations.

The *maintain* function may modify $plan_i$; further changes can also arise from receipt of *obligationMaintained* messages, sent by obligants upon changes (i.e., from local maintenances) to how delegated activities will be executed. Only the $i \in I$ (where I is the set of possible intentions) selected for execution is maintained; it is assumed i is selected on the basis of $goal_i$ (i.e., is goal driven behaviour), and that maintenance changes to $plan_i$ would not invalidate the basis for the original selection of i. This avoids unnecessarily maintaining the unselected members of I, as these other plans would be selected and executed under a future (almost certainly different) B – rendering reasoning over their *current* confidence unnecessary and likely inaccurate.

The *formAndUpdateContracts* function accounts for such changes by forming new contracts (if delegated activities have been added by maintenance), and communicating any changes to *existing* dependency or obligation contracts associated with a maintained i (for the former, relating to activities in the $plan_i$; for the latter, where $goal_i$ corresponds to an activity delegated from another). An *obligationMaintained* message, including the updated contract as part of the message body, is used to communicate changes in the latter case; this can result in sequential transmission of such messages as changes are propagated up the decompositional agent team.

Algorithm 1: The CAMP-BDI reasoning cycle; changes from the algorithm given by Rao and Georgeff (1995) are given by **bold** text

```
initializeState();
while agent is alive do
    D ← optionGenerator(eventQueue, I, B);
    i ← deliberate(D, I, B);
    /* (1) Maintenance of currently selected intention          i */
    if i ≠ Ø & i not waiting on a dependency to complete then
        i ← updateIntentions(D, I, B);
        B_i ← estimated execution context of i;
        maintain(i, B_i);
        formAndUpdateContracts(i);
        execute();
    /* (2) Maintenance of intentions in response to               */
    dependency changes received from obligant
    for each obligationMaintained message ∈ eventQueue do
        i_dependency ← the associated dependant intention;
        B_dependency ← estimated execution context of i_dependency;
        maintain(i_dependency, B_dependency);
        formAndUpdateContracts(i_dependency);
```

```
/* (3) Maintenance of obligants held by this agent, if */
no intentions were selected
if i = Ø then
    for each obligation contract ∈ agent's Obligations do
        i_ob_goal ← activity defined in obligation;
        i_ob_plan ← cached plan for obligation (to achieve i_ob_goal);
        i_ob ← i_ob_goal, i_ob_plan;
        B_ob ← execution context estimated using (causal links in obligation ∪ B);
        maintain(i_ob, B_ob);
        formAndUpdateContracts(i_ob);
getNewExternalEvents();
I dropSuccessfulAttitudes();
I dropImpossibleAttitudes();
I postIntentionStatus();
```

CAMP-BDI agents use *maintain* to identify whether any activities in a specified intention *i* (i.e., within the $plan_i$) are at risk of failure given the current *B*, and – if so – to perform mitigatory modification. The *maintain* algorithm (Algorithm 2) employs a two part process; first forming an ordered agenda (Algorithm 3) of *maintenance tasks* which each detail an activity under threat (see 4.1). The algorithm iterates through the resultant agenda, using *handleMaintenanceTask* (see 4.3) to consider and attempt the threat identified by a maintenance task, and terminating when either a task *has* been handled (i.e., threat addressed) or the entire agenda iterated through without success.

The *maintain* function will terminate after the first agenda task is successfully handled, as the associated to the $plan_i$ (through *handleMaintenanceTask*, described in 4.3) may invalidate the remaining agenda tasks. Although *maintain* could instead attempt to iteratively identify and handle maintenance tasks until no threats existed *or* none could be handled, this would risk uncertain termination conditions and higher computational cost. The decoupling of diagnosis and handling steps also facilitates future investigation towards improving either step.

Algorithm 2: The maintain function

```
Data: i – an intention; a plan plan_i to meet some goal goal_i
      B_i – The estimated execution context of the first activity in plan_i
handled ← false;
agenda ← formAgenda(goal_i; plan_i; B_i; empty agenda);
while ¬ handled and agenda ≠ do
    handled ← handleMaintenanceTask(agenda.removeTop());
Update Dependency contracts;
if i is an Obligation then
    Update contract and send to the dependant in an obligationMaintained message;
```

Figure 1 shows a motivating example where a *truck* intends (as part of its $plan_i$) to perform $a = move(M, F)$. The corresponding *move* capability defines, within the *pre* field, a requirement that the road being moved along must *not* to be flooded. When $M \rightarrow F$ *does* become flooded, *formAgenda* uses that *pre* field knowledge to identify the threat to a, inserting the corresponding maintenance task mt_a into the agenda. Following agenda generation, *handleMaintenance* task considers mt_a, and attempts to modify the $plan_i$ to prevent failure – such as to travel a different route or remove the flooding from $F \rightarrow M$ – based upon the capability set of that agent.

4.1 Maintenance tasks

A maintenance task (m_t) defines an activity under threat, and includes information relevant to the nature of that threat:

$$mt = \langle a, type, B_a, conf_a, mp_a \rangle$$

where a is an activity, intended to execute in the state given by B_a, which has an associated maintenance policy mp_a and has estimated confidence (given B_a) of $conf_a$. The *type* categorises both the type of threat and guides handling; this is defined as either *preconditions* or *effects*.

Maintenance task generation sees agents use capability knowledge to introspectively reason over their planned activities. A precedence ordering is applied if multiple capabilities may correspond to an activity; activities will be mapped to, in order, internal capabilities, contract-contained external capabilities, and advertised external capabilities (we assume agents will adopt the least complex approach for any activity, and avoid delegation if possible). Where multiple advertised external capabilities potentially apply, that with highest general confidence is mapped – under the assumption agents employ the same basis to arbitrate between obligant options.

Maintenance tasks in the agenda are ordered first by priority (defined through the relevant field of mp_a), and then a's position in the $plan_i$; i.e., the agenda will prioritise activities set to execute earlier unless others have been stated as higher priority in their maintenance policy.

The *Preconditions* type indicates a's preconditions do not hold in B_a. Successful handling sees the generation of a plan that will re-establish $pre(a)$, to be inserted into the $plan_i$ and executed prior to a (i.e., ensuring $pre(a) \not\subseteq B_a$). Preconditions handling effectively focuses upon ensuring the success of that specific individual a. However, the insertion of new activities (particularly if multiple preconditions tasks are being handled over multiple reasoning cycle iterations) can risk reducing the overall optimality of $plan_i$. Consequently, preconditions tasks are only generated where there is value in preserving that specific a in $plan_i$ – i.e., if a achieves a goal state or has an associated dependency contract (entailing communications costs from cancellation).

Effects maintenance tasks indicate that the threatened a can be *replaced* with some activity sequence which achieves the same effects, but with an increased level of confidence (given B_a). These tasks are generated where the estimated confidence of a is unacceptable (i.e., $conf_a < mp_a.threshold$), or if the preconditions of a do not hold in B_a and a does not require preservation. The latter condition helps prevent iterative $plan_i$ expansion, as might occur if preconditions type tasks were *always* generated where $pre(a) \not\subseteq B_a$.

4.2 Agenda formation

The *formAgenda* function (Algorithm 3) uses a recursive strategy (2) to support hierarchical plans (i.e., that involve activity decomposition), by iterating through leaf activities in their intended execution order. Leaf activities will typically be primitive, although they may also be (yet to be refined) composites in continuously planning agents. CAMP-BDI agents are assumed to possess capability knowledge (*know-how*) corresponding to every activity they can perform or decompose (i.e., regarding both any composites within the $plan_i$ and the activity signified by the $goal_i$ itself); the *getCapability* function identifies the appropriate capability object for a specified activity.

Once the appropriate capability (c_a) for a leaf activity *a* is identified, that c_a is used to determine whether a is at risk of failure – and if so, to guide generation and insertion of a new maintenance task within the agenda (1). In order to estimate the execution context for the subsequent leaf activity, B_a is updated with *a*'s (i.e., c_a. *eff*(*a*)) effects at the end of each iteration and returned; the updated Ba provides an estimated execution context for the following leaf activity during recursive *formAgenda* operations (but is discarded at the root *maintain* call level).

At the end of each iteration of *formAgenda*, the *consolidate* function (3) is called. If the agenda holds multiple maintenance tasks for activities within the same subplan, this generates and inserts a *single* effects maintenance task in replacement, where *mt.a* is the composite activity refined by that subplan. This merging behaviour is intended to avoid recurrent costs associated s individual re-diagnosis and handling of multiple individual threats, as would be incurred over sequential reasoning cycles. Instead, handling the consolidated maintenance task effectively resolves multiple individual threats within a single *maintain* operation, by seeking to reform the entire subplan (a minimal 'threatened' subset of $plan_i$) containing them.

4.3 Handling maintenance tasks

The *handleMaintenanceTask* function (Algorithm 4) 'handles' a given maintenance task (*mt*) by forming a *maintenance plan* ($plan_{fix}$) that, when inserted into the $plan_i$ (where $mt.a \in plan_i$), will mitigate the threat represented by *mt*. The specific sub-function employed depends upon *the* type of *mt*; *handlePreconditionsTask* [(1) – Algorithm 5] and *handleEffectsTask* [(3) – Algorithm 6) for *preconditions* and *effects* maintenance tasks respectively. Capability knowledge is employed to define both the goal and operator set (i.e., to determine what states must be established to address *mt*, and the activities the agent can use to do so) when specifying the maintenance planning problem (required to form $plan_{fix}$). Where a *preconditions* task cannot be handled, an equivalent field content *effects* task is generated and handled instead [(2)]; this relaxes the tighter planning constraints imposed by preconditions types (i.e., where *mt.a must* be preserved), allowing for replacement of *mt.a* rather than accepting its failure from the unmet preconditions. For example, a *Truck* unable to restore preconditions for *move*(*F*, *M*) would employ effects maintenance to find an alternate route (i.e., finding a $plan_{fix}$ that can replace *mt.a*) to achieve the required goal state of arriving *at*(*M*).

Algorithm 3: The formAgenda function

Data: g – a goal met, or composite activity performed, by p

p – plan of n activities $\{a_1,\ldots,a_n\}$ to perform g

agenda – priority ordered list of maintenance tasks; empty in initial (top-level) call

B_a – estimated execution context of a_0 in p

Result: *agenda* updated with maintenance tasks for p

B_a updated with post-effects of p (used by recursion)

B_{start}*leftarrow* copy of B_a (for execution context estimation);

for *each activity* $a \in p$ **do**

  if *a is abstract* **then**

    return *agenda*, B_a;

  $c_a \leftarrow$ getCapability(a);

  /* (1) Generate maintenance tasks for leaf activities */

  if c_a *primitive* || (c_a *composite* & *(a is not decomposed into a subplan))* **then**

    if *maintenance task mt found for leaf activity a* **then**

      Add *mt* to *agenda*;

      Update B_a with c_a.eff(a);

  /* (2) Recursion for decompositional subplans */

  else if c_a *composite* & *(a is decomposed into a subplan)* **then**

    $p_a \leftarrow$ subplan decomposing a;

    agenda, $B_a \leftarrow$ formAgenda(a, p_a, *agenda*, B_a);

  /* (3) Consolidate multiple tasks into one */

  agenda $\leftarrow$ consolidate(g, *agenda*, B_{start});

return *agenda*, B_a;

Algorithm 4: The *handleTask* function

Data: *mt* – a maintenance task

i – the intention requiring maintenance; $i = \{goal_i, plan_i\}$

Result: Boolean – true if $plan_i$ is modified and *mt* addressed.

handled $\leftarrow$ false;

if *mt.type* = *preconditions* **then**

  // (1) Handle preconditions type of *mt*

  handled $\leftarrow$ *handlePreconditionsTask*(*mt*, *i*);

  if $\neg$ *handled* **then**

    // (2) Create equivalent effects task for *mt.a*

    mt $\leftarrow$ new MaintenanceTask(*effects*, *mt.a*, $mt.B_a$, $mt.conf_a$);

else
return *handled*;

// (3) `Handle effects type of` *mt*
return *handleEffectsTask*(*mt*, *i*);

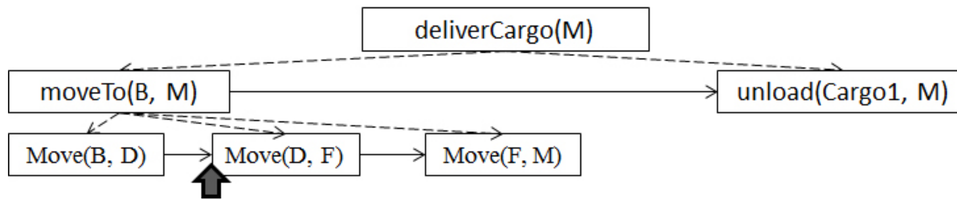
4.3.1 Performing preconditions maintenance

Preconditions task handling is performed by the *handlePreconditionsTask* function (Algorithm 5). The algorithm first attempts to generate a maintenance plan ($plan_{fix}$) to (re)establish *mt.a*'s preconditions, to be inserted prior to *mt.a* in $plan_i$ [similar to prefix plan repair in Komenda et al. (2014)]. To avoid requiring further maintenance, $plan_{fix}$ is only inserted if its estimated confidence exceeds $mt.mp_a.threshold$ – i.e., will not trigger generation of further effects maintenance tasks in the next reasoning cycle. This constraint is removed if *mt.a* is the next to execute in $plan_i$; our motivation assumes adoption of any non-zero confidence $plan_{fix}$ will be preferable to certain failure (and consequent debilitation) of an immediately executing *mt.a*.

Algorithm 5: The *handlePreconditionsTask* function

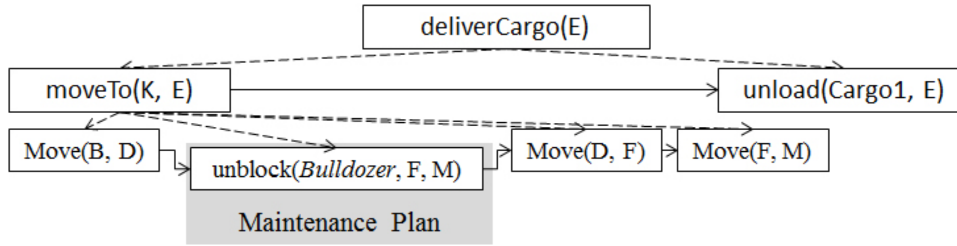
Data: *task* – a maintenance task
Result: true if a plan was found and inserted
$plan_{mt}$ ← plan containing *task.a*;
c_a ← getCapability(*task.a*);
Define planning problem $prob_a$, with initial state = $task.B_{mt}$ and goal = *ca*.pre(*task.a*);
if *plan* $plan_{fix}$ *solving* $prob_a$ *found* & $plan_{fix}$ *is acceptable* **then**
  Insert $plan_{fix}$ into $plan_{mt}$ as predecessor of *task.a*, and **return** true;
return false;

Figure 2 Example where *Truck*'s plan to *deliverCargo* is threatened by violated preconditions of *Move*(*D*, *F*), indicated by the arrow, following closure of *D* → *F*



An example of a preconditions maintenance scenario, deriving from our motivating example (Figure 1), is given in Figure 2. Here, *Truck* intends to travel to *M*; as *D* → *F* is blocked on the planned route, a new preconditions task is generated with *mt.a* = *move*(*D*, *F*) (we assume *Truck* wishes to preserve this activity, for the purposes of this example). Successful maintenance, given in Figure 3, sees generation of a maintenance plan, inserted and expected prior to *mt.a*, which employs a *Bulldozer* agent to reopen D → F (ensuring *mt.a*'s preconditions hold).

Figure 3 Example insertion of a successfully identified maintenance plan, restoring the preconditions of the threatened *Move*(*D*, *F*) through *unblock* clearing $D \rightarrow F$



4.3.2 Performing effects maintenance

Effects maintenance attempts to replace a (minimal) subset of the plan containing *mt.a*, with a maintenance plan that will achieve the same post-execution effects with superior confidence. Figure 2 shows an example scenario where plan preconditions hold, but *slippery* conditions on D → F have introduced risk which reduces confidence to an unacceptable level. This scenario would see generation of an effects maintenance task *mt* where *mt.a* = *move*(*D*, *F*); successful handling of *mt* sees a minimal subset of the $plan_i$ replaced with a maintenance plan that achieves the same outcome, with an acceptable level of confidence.

Effects maintenance uses an approach similar to HTN plan repair. Our algorithm (Algorithm 6) employs upwards recursion to iteratively (3 in the algorithm) re-refine composite activities (subgoals, or the $goal_i$), and terminates upon either forming and inserting an acceptable confidence maintenance plan ($plan_{fix}$ with confidence exceeding $mt.mp_a.threshold$) or upon attempting and failing to re-decompose the root $goal_i$.

Aside from the potential communications cost, a risk associated with dependency cancellation is that changes in circumstance may render external capabilities unavailable for *new* dependency contracts, even if a contract previously existed before being cancelled. This may stymie maintenance planning if a particular, now unusable, external capability is required for achieving $goal_i$ – although these risks also exist for post-failure replanning, and may apply to a greater degree than with maintenance (which attempts to modify only a *minimal* subset of the $plan_i$). By attempting to minimise changes to the $plan_i$, our algorithmic design trades-off the cost of potentially performing multiple planning operations against stability and computational costs associated with total replanning (Fox et al., 2006).

Algorithm 6: The *handleEffectsTask* function

Data: *mt* – a maintenance task

Result: **true** if a plan was found and inserted into the $plan_i$ containing *mt.a*

$a \leftarrow mt.a$;

$plan_{mt} \leftarrow$ intended plan containing *mt.a*;

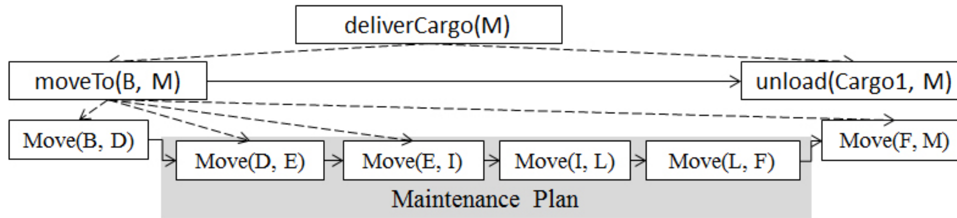
if $plan_{mt}$ *is a hierarchical plan* **then**

  $p_{mt} \leftarrow$ subplan of $plan_{mt}$ containing *a*;

```
else
   p_mt ← plan_mt;
B_mt ← mt.B_a;
/* (1) Attempt replacement of mt.a only                              */
if a not last in plan_mt || a has subsequent dependencies then
   ca ← getCapability(a);
   Define planning problem prob_a, with initial state = B_mt and goal = ca.effects(a);
   if plan plan_fix found for prob_a & plan_fix is acceptable then
      Replace a in p_mt with plan_fix;
      return true;
/* (2) Attempt replacement of mt.a and its suffix in p_mt             */
if a not first in plan_mt || a has preceding dependencies then
   a ← goal achieved by p_mt;
   c_a ← getCapability(a);
   Define planning problem prob_a, with initial state = B_mt and goal = c_a.effects(a);
   if plan_fix found for proba & plan_fix is acceptable then
      Replace the suffix of p_mt from a inclusive with plan_fix;
      return true;
/* (3) Iterates through increasingly abstract plan levels            */
while a ≠ root goal of plan_mt do
   a ← goal activity for p_mt;
   B_mt ← estimated execution context of a;
   c_a ← getCapability(a);
   Define planning problem prob_a, with initial state = B_mt and goal = c_a.effects(a);
   if plan_fix found for prob_a & plan_fix is acceptable then
      // (4) Use plan_fix to re-decompose/re-refine a
      Replace p_mt with plan_fix;
      return true;
return false;
```

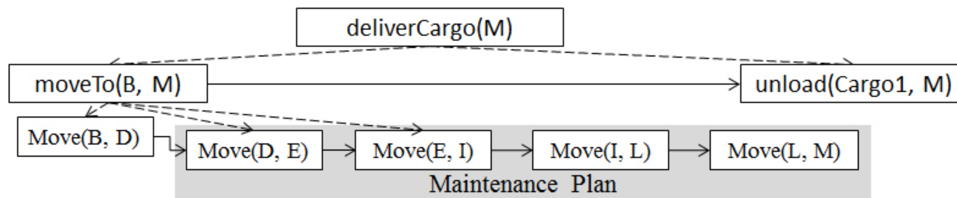
The algorithm attempts to minimise disruption to dependencies by first performing two restricted-scope planning operations at the most specific subplan level of iteration (i.e., the ‘lowest’ level subplan containing *mt.a* itself). If dependency contracts exist for *mt.a* or its successor activities (in that subplan), the agent first attempts to generate a maintenance plan ($plan_{fix}$) and replace *mt.a* only (1) – with the activities following *mt.a* (and associated dependency contracts) in the $plan_i$ being retained as the suffix of the newly inserted maintenance plan (Figure 4).

Figure 4 Example insertion of a maintenance plan as a substitute for *Move*(*D*, *F*), which achieves the same goal state (being at location *F*) from the estimated execution context (including start location) of *Move*(*D*, *F*)



If *mt.a* is *preceded* by dependencies (with existing contracts), the algorithm attempts *suffix* plan repair [similar to repeated lazy repair in Komenda et al. (2014)] – where the generated $plan_{fix}$ replaces *mt.a* and any succeeding activities in the subplan, but without altering the *predecessors* of *mt.a* (2). An example of this behaviour, where the inserted maintenance plan will achieves the same goal as the subplan whose suffix it replaces, is given in Figure 5. Both this and the prior direct substitution behaviour attempt to reduce disruption to a distributed plan by restricting the scope of possible changes, but also entail additional planner calls.

Figure 5 Example insertion of a successfully identified maintenance plan in the suffix case; the initial context of the threatened *Move*(*D*, *F*) is employed as the initial state for planning, with the goal defined as that of the parent *MoveTo*(*B*, *M*) activity



The algorithm has worst-case complexity equivalent to $O((n + 2)p)$ (where n is the number of plan levels and p the planning cost); i.e., where the algorithm attempts to plan at all levels of the hierarchical $plan_{mt}$, plus twice at the initial p_{mt} level (for a failed preconditions maintenance task, and for replacement of *mt.a* alone). This may still represent a significant *actual* computational cost, depending upon the computational cost of each planning operation.

5 Distributed behaviour

In a MAS, agents form teams to achieve goals which are impossible for individuals acting alone. The failure of an individual agent in such a team can have reciprocal impact upon other team members, risking failure of the distributed plan. Our design of distributed maintenance behaviour assumes activity delegation to, and decomposition by, obligants leads to hierarchical team structures. As the distribution of knowledge and capabilities in realistic domains frequently renders centralised approaches infeasible, this behaviour is designed to be decentralised; we use structured communication to drive the

adoption of maintenance responsibility at increasingly abstract levels of the agent team hierarchy (Figure 8), with use of the algorithms defined in Section 4.

Both internal and external capabilities share a common representation model, allowing use of the same maintenance algorithms to reason over both locally performed and delegated activities. The supporting architecture (Section 3) plays a critical role in distributed maintenance by providing external capability information *specific* to a delegated activity within associated dependency and obligation contracts. Local maintenance by dependant agents can utilise capability information specific to a delegated activity through the associated contract's *external capability* field; any semantic knowledge requirements are offset to the obligant(s), who will actually form that field's contents.

Following execution of *maintain* for an obligation $plan_i$ (i.e., whose $goal_i$ corresponds to an accepted obligation contract), the obligant transmits an *obligationMaintained* message to the (waiting, quiescent) dependant – this communicates an updated obligation contract, reflecting any modifications made to $plan_i$. Receipt of *obligationMaintained* informs the dependant that the obligant has made any confidence-raising changes possible through local maintenance – allowing the dependant to adopt responsibility and maintain its own dependant intention, with the awareness that the obligant has already made any changes it can. Obligant maintenance is performed if the agent is either executing an (intention corresponding to an) obligation, or does not hold *any* intentions (Algorithm 1) – in the latter case, otherwise idle agents will act to maintain mutual beliefs (i.e., update the associated contract) regarding accepted obligations.

A dependant will adopt maintenance responsibility if and when an obligant cannot maintain sufficient confidence in, or ensure preconditions hold for, its obligation – i.e., the subpart of the distributed plan the obligant intends to execute. Responsibility is gradually adopted 'up' the distributed team hierarchy until an agent has performed maintenance with an outcome acceptable to both itself and any direct dependant (of the maintained intention $goal_i$) – for the latter, the obligation contract (external capability) must indicate preconditions hold and sufficient confidence is held in the obligation, such that no maintenance tasks would be generated when considering that delegated activity within the dependant $plan_i$. The resulting upwards escalation restricts distributed plan changes to the most specific ('lowest') agent level possible; Figure 8 shows an example of this behaviour, also described below:

1 Agents *C* and *D* call *maintain* within their local reasoning cycle(s).

2 *C* and *D* individually perform post-maintenance messaging, with each sending a *obligationMaintained* message to *B* (including contracts updated to account for maintenance changes).

3 *B* calls *maintain* locally upon receipt of *obligationMaintained* messages from all obligants. The messaged, updated contracts are used to update the contract held by *B* for that dependency.

4 Upon *B* completing execution of *maintain*, the further updated contract (accounting for both the changes from *C* and *D* and the outcome of *B*'s *maintain* operation) is communicated to *A* within an *obligationMaintained* message.

5 *A* calls *maintain* upon receiving *B*'s message. *A*'s plan does not correspond to an obligation, so no further post-*maintain* messaging is required.

We synchronise distributed maintenance by using contracts to define a maintenance policy, and specifically confidence thresholds, common to the obligant(s) and dependant involved in a delegated activity. This, combined with the sequencing of *maintain* calls and *obligationMaintained* messages (i.e., contract updates), ensures a dependant will only generate an effects maintenance task for a delegated activity under conditions where the obligant must have already done the same. Whilst the preceding example depicts linear dependency formation, it is possible for indirect 'self dependencies' to occur – e.g., if *D*'s $plan_i$ includes a dependency upon some *other* capability of *A*.

Figure 6 Example of a distributed intention, where *Truck1* holds an obligation to perform the two activities *moveTo*(*B*, *M*) and *unload*(*Cargo1*, *M*)

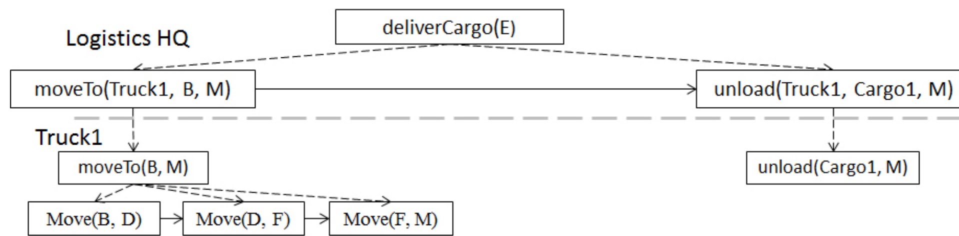
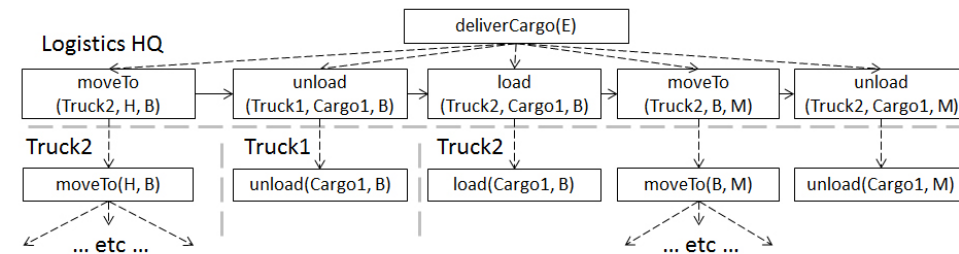


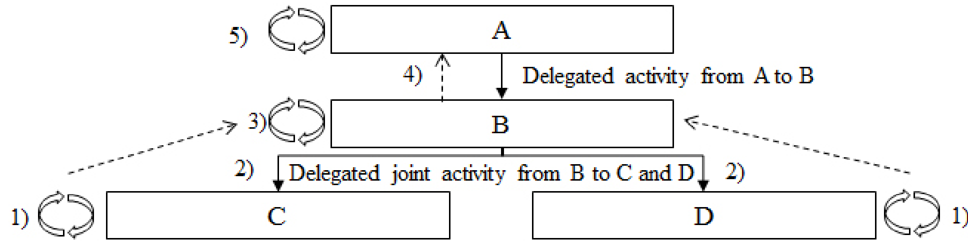
Figure 6 illustrates an example of maintaining a distributed intention; here, damage to *Truck1* has reduced confidence in its obligation to perform a delivery task for *LogisticsHQ*. As *Truck1* lacks the capability to repair itself, it cannot restore confidence above the relevant maintenance policy threshold. Following receipt of *Truck1*'s *obligationMaintained* message (conveying this confidence change and signifying *Truck1* cannot offer any improvement), subsequent maintenance of the dependant intention by *LogisticsHQ* identifies insufficient confidence in the corresponding activity. The modifications from successfully handling the resulting effects maintenance task see *LogisticsHQ* use an alternate, undamaged obligant (*Truck2*), which offers superior confidence over *Truck1* (Figure 7). This distributed maintenance behaviour is equivalent to maintaining a *local* hierarchical $plan_i$, but is performed by a hierarchical team where the distributed plan is being effectively refined through delegation of activities to obligants.

Figure 7 Result of adoption of maintenance responsibility by *Logistics HQ* in response to low confidence in *Truck1*'s obligation (Figure 6)



Notes: The new obligant *Truck2* originates at a different initial location and must first travel to *B* to retrieve *Cargo1* – which was previously carried by, and now must be unloaded by, *Truck1*.

Figure 8 The adoption of responsibility process in a hierarchical team, where *B* is an obligant of *A*, and *C* and *D* are obligants for a joint activity in B's plan



Our distributed maintenance behaviour seeks to replicate re-refinement HTN plan repair, but for a distributed plan where dependency relationships (namely the resultant identification of plans by obligants to *perform* obligated activities) are analogous to task refinement. Agents adopt responsibility for maintenance when executing planned activities, or upon an obligant informing them that they have performed maintenance (Algorithm 1). The latter scenario sees updated dependency contract information used by the dependant to determine if obligant maintenance was successful, and – through considering changes against conditions defined by the contract maintenance policy – whether local modification is required for the dependant $plan_i$.

6 Evaluation

Our evaluation compared a CAMP-BDI MAS against one using a reactive failure mitigation approach, operating within a logistics domain corresponding to our motivating example (described in Section 2). Agents within the reactive MAS attempted replanning upon activity failure; we argue this is an appropriate method for handling unexpected outcomes, as it represents a conceptually similar approach to that employed by leading determinisation-based probabilistic planners. For example, *FF-Replan* (Yoon et al., 2007) takes advantage of historical optimisations to classical planning by determinising a probabilistic domain; differences between the actual and anticipated (classical operator modelled) outcome are detected through plan execution monitoring (PEM) and trigger reactive replanning. The reactive MAS used a single-outcome determinisation, with success regarded as the most probable outcome if preconditions held; activity failure was treated as an unexpected outcome and triggered replanning (mirroring *FF-Replan*). Baseline 'worst-case' performance was determined with a MAS where agents employed *no* failure mitigation strategy.

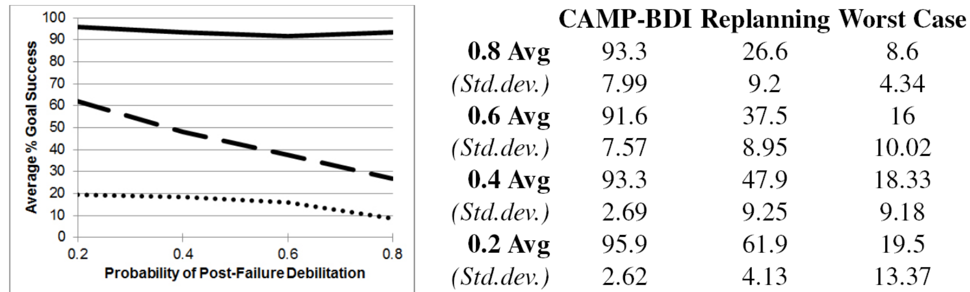
All experiments were performed on a system with an Intel i5-3750k processor (3.5Ghz) and 16GB RAM, running Java v1.8.0 31; MASs were implemented by extending the *Jason* agent framework (Bordini and Hübner, 2006) to support contract formation as part of distributed intention execution and to support runtime planning within CAMP-BDI and replanning agents. We adopted *LPG-td* (Gerevini and Serina, 2002) as a planner. We deemed use of a classical planner as an appropriate analogue to a real-world implementation, due to offering both superior flexibility over HTN or plan library methods and faster performance than probabilistic approaches.

MAS performance was compared in terms of three metrics; overall goal achievement (total number of successful deliveries), the number of activities executed per goal

achieved (efficiency), and the average planning calls per achieved goal (as an indicator of computational cost). A variety of exogenous changes could occur in the environment; landslips (blocking roads), locations becoming dangerous, or rainfall leading to slippery and then flooded roads (leading, respectively, to increased risk of and then guaranteed failure for activities requiring use of said road). Each MAS was formed of heterogenous agents able to achieve system goals through delegation and co-ordination. *Truck* agents capable of load, unload and movement activities necessary for transport of cargo objects, with a variety of other agent types providing capabilities for addressing negative world states; APCs are used to render dangerous areas safe, *Hazmats* to decontaminate toxic roads, and *Bulldozers* to clear roads blocked by landslips.

Performance of each approach was evaluated for, and averaged over, ten runs; each run lasted for (generation, and success or failure in achieving) 100 cargo delivery goals. All experiments employed the same procedurally generated geography, using a specified seed value to control simulation events. Activity failure risked various types of debilitation including damage to the agent (of incremental severity and with associated confidence loss) and – if the agent was loaded with, loading or unloading cargo – destruction or spillage of cargo (the latter applied only where cargo was of a *hazardous* type, and would render roads toxic). Agent damage was gradually recovered from (‘healed’) whilst that agent was idle. Performance of each system was evaluated for 20, 40, 60 and 80% (n = 0.2, 0.4, 0.6 and 0.8) probabilities of the above debilitations arising following failure; probabilities were applied individually to each type of debilitation.

Figure 9 Average goal achievement rate (%) for 0.2 to 0.8 post-failure damage probability, with standard deviation



	CAMP-BDI	Replanning	Worst Case
0.8 Avg	93.3	26.6	8.6
(Std.dev.)	7.99	9.2	4.34
0.6 Avg	91.6	37.5	16
(Std.dev.)	7.57	8.95	10.02
0.4 Avg	93.3	47.9	18.33
(Std.dev.)	2.69	9.25	9.18
0.2 Avg	95.9	61.9	19.5
(Std.dev.)	2.62	4.13	13.37

Notes: CAMP-BDI results are shown as solid lines, replanning dashed, and worst-case as dotted.

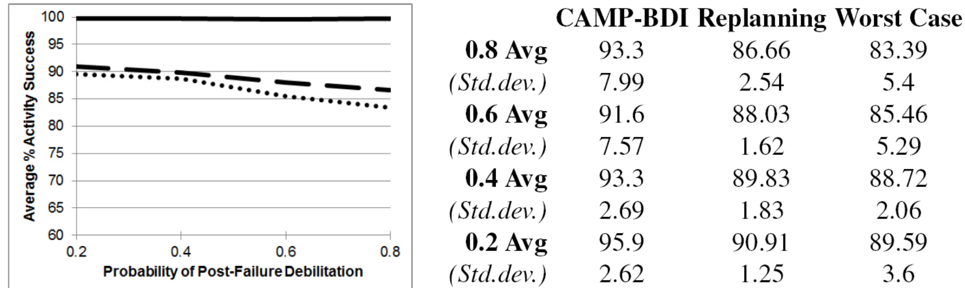
Figure 9 shows that CAMP-BDI achieved significantly more goals than replanning, with increasing superiority as n increased. CAMP-BDI achieved around 95% of goals for all probabilities of debilitation; replanning achieved 61.9% of goals at n = 0.2, and eventually dropped to 26.6% at n = 0.8. Worst-case performance was consistently poor; from n = 0.2 to 0.6 19.5% to 16% of goals were achieved, with this falling to 8.6% as post-failure debilitation became virtually certain at n = 0.8. Debilitation had a reduced impact at lower n in the worst-case system as agents would fail goals immediately upon any activity failure, regardless of whether debilitation had occurred. Replanning, in contrast, would generally only fail a goal upon repeated activity failure and reactive replanning; this risked accumulating debilitation(s) from these individual activity failures, and increased the overall risk of at least *one* debilitation occurring during the pursuit of

$goal_i$. Replanning still offered superior performance over the worst-case, as certain (debilitated or otherwise) post-failure states could still be recovered from.

The increasing superiority of CAMP-BDI over replanning was attributed to the increasing risks of post-failure debilitation; damage also persisted to impact subsequent activities, giving successive failures a compounding debilitative effect. CAMP-BDI's proactivity offered two advantages. Firstly, preventing failure avoids agents having to plan and act in debiliated (suboptimal) post-failure states. The confidence loss associated with agent damage also saw maintaining dependants seek to use higher-confidence (i.e., undamaged) obligants, reducing the workload upon agents with suboptimal health – avoiding compounding damage from any subsequent failure of that obligant (e.g., *due* to existing damage) and allowing health recovery when idle. We can also imagine an extension to our experimental environment where agents recovered health using explicit repair activities rather than through 'passive' repair (i.e., whilst idle). In such cases, CAMP-BDI could stimulate such repair upon detecting the confidence loss from damage – while reactive approaches can only respond *after* consequent failure.

The worst-case MAS generally had less pronounced decreases in goal achievement as *n* increased; we judged this as being due to *immediate* failure, whilst replanning would only ultimately fail from the cumulative effect of *repeated* debilitation from activity failure-then replanning cycles. At *n* = 0.8, post-failure debilitation became almost certain; this was reflected through a notable drop in worst-case performance between *n* = 0.6 and 0.8 (Figure 9).

Figure 10 Average activity success (%), for 0.2 to 0.8 post-failure damage probability, with standard deviation



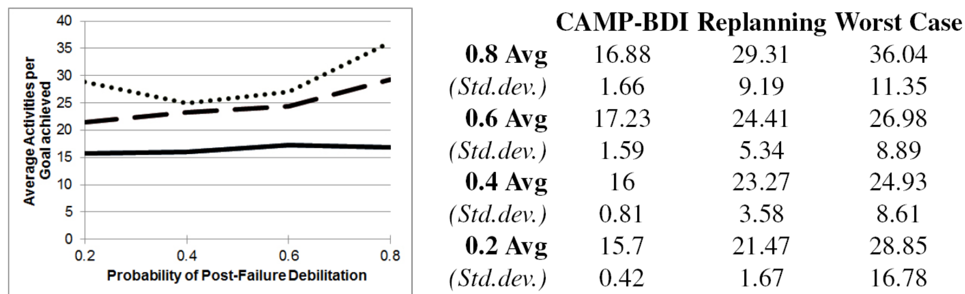
	CAMP-BDI	Replanning	Worst Case
0.8 Avg	93.3	86.66	83.39
(Std.dev.)	7.99	2.54	5.4
0.6 Avg	91.6	88.03	85.46
(Std.dev.)	7.57	1.62	5.29
0.4 Avg	93.3	89.83	88.72
(Std.dev.)	2.69	1.83	2.06
0.2 Avg	95.9	90.91	89.59
(Std.dev.)	2.62	1.25	3.6

Notes: CAMP-BDI results are shown as solid lines, replanning dashed, and worst-case as dotted.

Figure 10 shows a similar pattern for average activity success, with CAMP-BDI maintaining consistent success rates (99.78% to 99.70%) from *n* = 0.2 to 0.8, whilst both replanning (90.90% to 86.66%) and worst-case (89.59% to 83.39%) systems saw activity failures increase with greater values of *n*. This would seem intuitive, given that CAMP-BDI focuses upon proactive avoidance of failure versus reactive response (or ignorance, as in the worst-case); our results confirmed CAMP-BDI was effective at preventing activity failure. For all systems, including the worst-case, activity success rates were generally high, as failures typically followed successful execution of multiple preceding activities. Decreases in activity success rates for replanning and worst-case systems can be attributed to the increasing likelihood of post-failure debilitation, particularly as negative states could persist over time.

One potential concern for CAMP-BDI was the potential *cost* of employing runtime planning within a proactive approach; as Toyama and Hager (1997) note, reactive failure handling offers the benefit of only expending recovery costs after a definitive failure, rather than upon anticipating *potential* failure. To this effect, our results (Figure 12) did show CAMP-BDI as executing significantly more planning operations per goal at lower n values than replanning (e.g., 9.91 versus 5.642 planning calls per achieved goal at $n = 2$). However, replanning became significantly less efficient as n increased and debilitation became more likely; at $n = 0.8$, replanning performed an average 19.91 calls per achieved goal, compared to 11.03 for CAMP-BDI. This reflected an increasing risk of post-failure debilitation (including scenarios where reactive recovery was rendered impossible), and suggests the costs of employing a proactive approach may be balanced against the costs of 'allowing' failure before invoking a recovery strategy. The higher costs of proactivity may be further justifiable if failure can have particularly severe consequences – e.g., if destroyed cargo was nuclear waste or vital medical supplies.

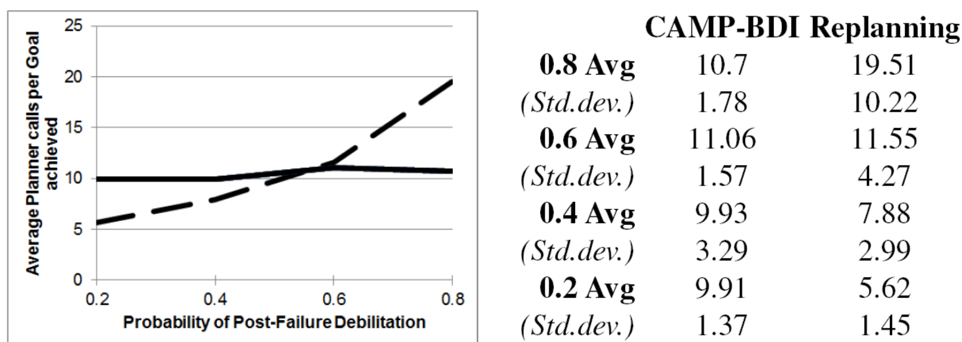
Figure 11 Average activities per goal, for 0.2 to 0.8 post-failure damage probability, with standard deviation



	CAMP-BDI	Replanning	Worst Case
0.8 Avg	16.88	29.31	36.04
(Std.dev.)	1.66	9.19	11.35
0.6 Avg	17.23	24.41	26.98
(Std.dev.)	1.59	5.34	8.89
0.4 Avg	16	23.27	24.93
(Std.dev.)	0.81	3.58	8.61
0.2 Avg	15.7	21.47	28.85
(Std.dev.)	0.42	1.67	16.78

Notes: CAMP-BDI results are shown as solid lines, replanning dashed, and worst-case as dotted.

Figure 12 Average planner calls per goal achieved for 0.2 to 0.8 post-failure damage probability, with standard deviation



	CAMP-BDI	Replanning
0.8 Avg	10.7	19.51
(Std.dev.)	1.78	10.22
0.6 Avg	11.06	11.55
(Std.dev.)	1.57	4.27
0.4 Avg	9.93	7.88
(Std.dev.)	3.29	2.99
0.2 Avg	9.91	5.62
(Std.dev.)	1.37	1.45

Notes: CAMP-BDI results are shown as solid lines and replanning as dashed.

One possible CAMP-BDI optimisation is to include temporal considerations within maintenance – for example, limiting (in configuration or through maintenance policies) the number of activities into the future (how 'deep' into the $plan_i$) maintenance should

consider. This could help balance the costs of maintaining more temporally distant activities against the benefits of earlier anticipation and response to failure risks. Such threshold configurations would likely depend upon the domain, considering factors such as average plan length, or the likelihood of various types of exogenous change over time – the latter reflecting the degree of uncertainty over whether the estimated execution context (used to anticipate failure) for an activity would have held at execution time.

The average number of activities executed per each goal *achieved* (Figure 11) served as an indicator of the *cost* of each goal achieved. CAMP-BDI shown fairly consistent performance for this metric, executing an average 15.69 to 16.88 activities per goal achieved (from n = 0.2 to 0.8). In contrast, replanning shown increasing activity cost, from 21.47 activities executed per goal at n = 0.28 to 29.31 at n = 0.8. This reflected the increasing difficulty of the environment (greater n entailed more frequent debilitation, with consequently more frequent confidence loss, activity failure and replanning), particularly as less goals were achieved – the absolute total of activities (as averaged across all experimental runs) executed by replanning agents actually decreased from 1,322.99 at n = 0.2 to 700.3 at n = 0.8. CAMP-BDI, in contrast, was relatively consistent; ranging from 1,504.6 activities at n = 0.2 to 1,564.2 at n = 0.8.

Results in the worst-case system were more variable; the average activities per goal actually *decreased* between n = 0.2 and 0.4, from 28.85 to 24.93, before rising to a maximum of 39.04 at n = 0.8. This variation may be due to an extremely low goal success rate in the worst-case system, combined with variations in exactly *when* execution of the intention failed. While a comparatively modest increase is shown for n = 0.6, we note that one worst-case experimental run was discounted from the total average after failing to achieve *any* goals; this prevented calculation of an average activity cost, and suggests the overall averaged activities-per-goal would be much higher if this zero goal run had somehow been factored in. Due to the extremely low overall rates of goal achievement, the worst-case system values for this metric may not necessarily indicate the activity cost of goals, but rather at what point during distributed plan execution failure actually occurred – this point may have been more random than for CAMP-BDI and replanning systems, which would still *attempt* to respond to potential or actual failures.

Our results, in summary, show a clear advantage for CAMP-BDI where activity failure risks debilitative consequences. While proactivity may risk additional costs from false-positive anticipation of failure, our results suggest this may be mitigated where the *consequences* of 'allowing' (and only *reacting* to) activity failure are likely to stymie reactive recovery, or hinder subsequent goals. However, we must note it is infeasible to expect any proactive approach to anticipate and prevent *every* failure in every realistic environment, particularly where exogenous change can occur during (and ergo fail) activity execution.

As a consequence, it is likely *some* form of reactive failure recovery will always be required to handle random, unpredictable failures. We suggest CAMP-BDI offers a complimentary approach to reactive approaches, which can be targeted at those activity types whose failures *are* potentially preventable and which risk consequences that would hinder recovery. Our use of maintenance policies also provides the potential for optimisation; for example, confidence thresholds could be reduced for activity types with lower risks of post-failure debilitation, allowing use of reactive failure handling instead. This would reduce iterative costs of maintenance, under an assumption it would be possible to recover from any failure of that activity type.

7 Related work

A variety of existing work influenced and inspired CAMP-BDI. The capability model draws from concepts of *know-how-to-perform*, *can-perform* and *know-how-to-achieve* by Morgenstern (1986) and work by Singh (1999) regarding *know-how*. Plan confidence estimation resembles TÆMS quality metrics (Lesser et al., 2004), specifically q_min (future work may investigate further alternatives). He and Ioerger (2003) also discuss quantitative quality estimation, but to maximise schedule efficiency. The potential value of capability knowledge within BDI reasoning has also been explored; Sabatucci et al. (2013) use capabilities to representing plans and their viability conditions to evaluate the achievability of desires during intention selection. Waters et al. (2014) propose an approach towards intention selection which favours selection of the most *constrained* options, using knowledge of selection preconditions to prioritise selectable plans with the lowest coverage (Thangarajah et al., 2012). Unlike CAMP-BDI, they sought to maximise overall intention throughput rather than ensure success of the current intended goal – although our capability model can support similar reasoning for desire and intention selection, beyond our current robustness focus.

CAMP-BDI shares some conceptual similarities with both PEM – e.g., SIPE (Wilkins, 1983) – and plan repair – e.g., O-Plan (Drabble et al., 1997) – approaches, as all identify or respond (through reformation or modification) to divergence from expected states during plan execution. Plan repair can offer benefits over replanning for *distributed* plans, by minimising the scope of plan changes (i.e., maximising plan stability) and consequently reducing the cost of communicating such changes to others. However, CAMP-BDI differs from these approaches by explicitly focusing upon BDI agent reasoning and by extending agent-level maintenance behaviour to the distributed case; our use of confidence estimation also provides a qualitative aspect to this response behaviour, whilst PEM uses deterministic preconditions to diagnose if expected and actual states have diverged.

Braubach et al. (2006) defines goal types driving agent proactivity as being to either *achieve* or *maintain* a state, over some defined period or while defined conditions hold; *reactive* (to re-establish the state if violated) and *proactive* (constraining plan and goal adoption to prevent state violation) subtypes of the latter are defined by Duff et al. (2006). Reactive maintenance goals stimulate achievement goals to restore the protected states if and when violated; the subsequent intentions can be maintained through CAMP-BDI. Identification and handling of preconditions maintenance tasks in CAMP-BDI is similar to inferring proactive maintenance goals to preserve required precondition states until execution of the relevant activity. Effects maintenance also bears similarities, as loss of high-confidence associated states similarly triggers plan modification – although maintenance planning may identify an *alternate* sequence of activities, rather than being solely bound to restore said high confidence states. We assume that the methods used by agents to determine plans for intentions and form maintenance plans will recognise and observe any defined maintenance goals.

Hindriks and Van Riemsdijk (2007) suggest an approach which, like CAMP-BDI, employs (limited) lookahead. Rather than ensure successful activity execution, they use this lookahead to ensure proactive maintenance goals are respected; their approach forms and uses goal-plan trees to anticipate the future effects of plans, and consequently to prevent selection (intention) of plans which would violate such goals. Unlike the CAMP-BDI view of plans as modifiable, they treat plans as pre-defined and immutable –

with (anticipated) state violation suggested as best addressed through relaxing goals to allow adoption of alternate plans. However, such relaxation may not be viable in certain domains or scenarios – for example, if goals correspond to *safety responsibilities* (Wooldridge et al., 2000) where the agent must ensure particular critical states hold. A similar approach, again using a goal-plan tree to guide goal adoption based on plan effects, is also suggested by Duff et al. (2006). CAMP-BDI varies from these approaches by also considering exogenous change as potential sources of violation, and by focusing upon preventing failure of *existing* intentions – whilst proactive maintenance goals instead constrain the selection of desires and formation of intentions to preserve states.

Proactive maintenance goals *can* impact runtime activity by constraining subgoal refinement by agents employing continual planning. Continual planning strategies handle uncertainty through deferring the refinement of certain abstract activities until plan execution, at a timepoint where knowledge (of their execution context) is considered more certain (des Jardins et al., 1999). CAMP-BDI uses composite capabilities to support such an approach; this type allows analysis of whether subgoals can be decomposed, through both representing the plan options (and precondition constraints) for refining a given activity and by providing confidence estimation (i.e., an indicator of quality based on such options). If sensing activities exist (i.e., to learn specific information before runtime refinement of activities), the knowledge required for and gained from execution can be represented within capability preconditions and effects.

An alternate approach for acting within stochastic domains is offered by Markov decision processes (MDPs); MDPs use state transition probabilities and a reward function to generate a *policy* defining an optimal activity for each possible world state. Partially Observable MDPs remove the total knowledge requirement of MDPs by defining a probability map of state *observations* – this is used to infer the actual state and define a solvable MDP. Although the resultant policies offer optimal behaviour, complexity issues can render MDP approaches intractable for realistic environments as state space increases; applicability can be improved through state space abstraction, but this also loses overall policy optimality (Boutilier and Dearden, 1994).

BDI approaches can be regarded as a more efficient alternative to MDPs; Schut et al. (2002) show BDI agents as able to handle domains intractable for MDPs, whilst offering approximate performance (depending upon runtime planning costs). Work has also sought to reconcile BDI with MDP approaches; work by Simari and Parsons (2006) identifies similarities and potential mappings between policies and plans. Subsequent extension by Pereira et al. (2008) defined an algorithm to form deterministic plans (for use in agent libraries) from POMDPs – but assumed POMDP policies could be formed offline, which may not be feasible for complex domains. MDP specifications can also be unintuitive, limiting their usability; Meneguzzi et al. (2011) suggest a method to map more intelligible HTN domains to MDP equivalents, defining transition probabilities based upon states within operation preconditions rather than environmental probabilities.

In our design of CAMP-BDI, we assumed deterministic plans (and operator specifications) are required for realistic domains due to the likely intractability of MDP approaches. Confidence estimation provides information similar to an MDP transition function, but provides a scalar *estimate* (providing flexibility for implementation) rather than requiring an exact probability.

8 Conclusions

We have contributed the CAMP-BDI approach towards distributed plan execution robustness, describing both an algorithm for performing pre-emptive plan modification (*maintenance*) and the use of structured messaging to extend local maintenance behaviour towards the distributed case. We also describe provision of supporting capability and contract meta-knowledge, and the use of maintenance policies to tailor this behaviour during runtime. Whilst CAMP-BDI – or any proactive approach – cannot wholly replace reactive methods (some failures will inevitably be impossible to anticipate or prevent), we argue it offers a valuable complementary approach.

Our approach does incur analysis costs when forming capability knowledge, which must be balanced against the risk (costs) of post-failure debilitation. These analysis requirements may be justified by a need for similar information to specify library plans or planning domains, which require similar understanding of how and which environment states impact activity outcomes. Specification costs are somewhat reduced through only requiring an indicative value from confidence estimation; this allows implementation granularity to be tailored to reflect the available (or discoverable) domain knowledge. The *benefit* (value) of modelling capability knowledge is arguably increased by their potential application within other robustness approaches, or to aid desire and intention selection.

Future work can explore minimisation of planning costs through expanded use of maintenance policies; e.g., introducing fields that define permissible relaxations for maintenance planning, or conditional rules indicating where maintenance is considered intractable (allowing responsibility to be deferred to dependants, avoiding 'futile' local planning effort). Further extension could define conditional rules that allow selection of pre-formed plan recipes for specific cases, used to avoid runtime planning. As CAMP-BDI does not mandate any specific planner implementation, use of heterogeneous planning approaches may be examined; e.g., with more specific (lower level) agents using fast HTN or library methods to improve reactive speed, but higher-level (abstract, broker or logical organiser) agents using more costly – but flexible – classical planning when the former cannot restore confidence locally. Greater optimisation of confidence estimation and agenda formation is also possible, although the most effective approaches may prove domain-specific.

Acknowledgements

This work was funded with support from EADS Innovation Works. Alan White would like to extend additional thanks to Dr. Stephen Potter for his invaluable help and advice. The authors and project partners are authorised to reproduce and distribute reprints and online copies for their purposes, notwithstanding any copyright annotation hereon.

References

Bordini, R.H. and Hübner, J.F. (2006) 'BDI agent programming in AgentSpeak using Jason', in Toni, F. and Torroni, P. (Eds): *Computational Logic in Multi-Agent Systems*, *Lecture Notes in Computer Science*, Vol. 3900, pp.143–164, Springer Berlin Heidelberg.

Boutilier, C. and Dearden, R. (1994) 'Using abstractions for decision theoretic planning with time constraints', *Proceedings of the 12th National Conference on Artificial Intelligence*, pp.1016–1022, Morgan Kaufmann, San Francisco, CA.

Braubach, L., Pokahr, A. and Lamersdorf, W. (2006) 'Extending the capability concept for flexible BDI agent modularization', in Bordini, R.H., Dastani, M.M., Dix, J. and Seghrouchni, E.F. (Eds.): *Programming Multi-Agent Systems*, *Lecture Notes in Computer Science*, Vol. 3862, pp.139–155, Springer Berlin Heidelberg.

des Jardins, M.E., Durfee, E.H., Ortiz Jr., C.L. and Wolverton, M.J. (1999) 'A survey of research in distributed, continual planning', *The AI Magazine*, Vol. 20, No. 4, pp.13–22.

Drabble, B., Dalton, J. and Tate, A. (1997) 'Repairing plans on-the-fly', *Proceedings of the NASA Workshop on Planning and Scheduling for Space*.

Duff, S., Harland, J. and Thangarajah, J. (2006) 'On proactivity and maintenance goals', *AAMAS-06*, pp.1033–1040.

Fox, M., Gerevini, A., Long, D. and Serina, I. (2006) 'Plan stability: replanning versus plan repair', *Proc. ICAPS*, pp.212–221, AAAI Press.

Gerevini, A. and Serina, I. (2002) 'LPG: a planner based on local search for planning graphs', *Proc. of 6th Int. Conf. on AI Planning Systems (AIPS'02)*, AAAI Press.

He, L. and Ioerger, T.R. (2003) 'A quantitative model of capabilities in multi-agent systems', In *Proceedings of the International Conference on Artificial Intelligence, IC-AI '03*, Las Vegas, Nevada, USA, 23–26 June, Vol. 2, pp.730–736.

Hindriks, K.V. and Van Riemsdijk, M.B. (2007) 'Satisfying maintenance goals', *Proc. of DALT'07*, Springer.

Komenda, A., Novak, P.and Pechoucek, M. (2014) 'Domain-independent multi-agent plan repair', *J. Network and Computer Applications*, Vol. 37, pp.76–88 [online] http://www.sciencedirect.com/science/journal/10848045/37?sdc=1.

Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., Nagendra Prasad, M., Raja, A., Vincent, R., Xuan, P. and Zhang, X.Q. (2004) 'Evolution of the GPGP/TÆMS domain-independent coordination framework', *Autonomous Agents and Multi-Agent Systems*, Vol. 9, Nos. 1–2, pp.87–143.

McCarthy, J. (1958) 'Programs with common sense', *Proceedings of the Teddington Conference on the Mechanisation of Thought Processes*, pp.77–84.

Meneguzzi, F., Tang, Y., Sycara, K. and Parsons, S. (2011) 'An approach to generate MDPs using HTN representations', *Decision Making in Partially Observable, Uncertain Worlds: Exploring Insights from Multiple Communities*, Barcelona, Spain.

Morgenstern, L. (1986) 'A first order theory of planning, knowledge, and action', *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge, TARK '86*, San Francisco, CA, USA, pp.99–114, Morgan Kaufmann Publishers Inc.

Pereira, D.R., Goncalves, L.V., Dimuro, G.P. and Costa, A.C.R. (2008) 'Constructing BDI plans from optimal POMDP policies, with an application to AgentSpeak programming', *Proc. of Conf. Latinoamerica de Inform´atica, CLEI*, Vol. 8, pp.240–249.

Rao, A.S. and Georgeff, M.P. (1995) 'BDI agents: from theory to practice', *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pp.312–319.

Sabatucci, L., Cossentino, M., Lodato, C., Lopes, S. and Seidita, V. (2013) 'A possible approach for implementing self-awareness in JASON', *EUMAS'13*, pp.68–81.

Schut, M., Wooldridge, M. and Parsons, S. (2002) 'On partially observable MDPs and BDI models', Selected Papers from the *UKMAS Workshop on Foundations and Applications of Multi-Agent Systems*, London, UK, UK, pp.243–260, Springer-Verlag.

Simari, G.I. and Parsons, S. (2006) 'On the relationship between MDPs and the BDI architecture', *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '06*, New York, NY, USA, pp.1041–1048, ACM.

Singh, D., Sardinia, S., Padgham, L. and Airiau, S. (2010) 'Learning context conditions for BDI plan selection', *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1, AAMAS '10*, Richland, SC, pp.325–332, International Foundation for Autonomous Agents and Multiagent Systems.

Singh, M.P. (1999) 'Know-how', in Wooldridge, M. and Rao, A. (Eds): *Foundations of Rational Agency*, *Applied Logic Series*, Vol. 14, pp.105–132, Springer, Netherlands.

Thangarajah, J., Padgham, L. and Winikoff, M. (2003) 'Detecting and avoiding interference between goals in intelligent agents', *IJCAI-03*, pp.721–726.

Thangarajah, J., Sardina, S. and Padgham, L. (2012) 'Measuring plan coverage and overlap for agent reasoning', *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems – Volume 2, AAMAS '12*, Richland, SC, pp.1049–1056, International Foundation for Autonomous Agents and Multiagent Systems.

Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N. and Uszok, A. (2003) 'Semantic web languages for policy representation and reasoning: a comparison of KAoS, Rei, and Ponder', in Fensel, D., Sycara, K. and Mylopoulos, J. (Eds.): *The Semantic Web – ISWC 2003*, *Lecture Notes in Computer Science*, Vol. 2870, pp.419–437, Springer Berlin Heidelberg.

Toyama, K. and Hager, G. (1997) 'If at first you don't succeed…', *Proc. AAAI*, Providence, RI, pp.3–9.

Waters, M., Padgham, L. and Sardina, S. (2014) 'Evaluating coverage based intention selection', in *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, Paris, France, May, pp.957–964, IFAAMAS, Nominated for Jodi Best Student Paper award.

Wilkins, D.E. (1983) 'Representation in a domain-independent planner', *Proceedings of the 8th International Joint Conference on Artificial Intelligence*. Karlsruhe, FRG, August, pp.733–740.

Wooldridge, M., Jennings, N.R. and Kinny, D. (2000) 'The Gaia methodology for agent-oriented analysis and design', *Autonomous Agents and Multi-Agent Systems*, September, Vol. 3, No. 3, pp.285–312.

Yoon, S.W., Fern, A. and Givan, R. (2007) 'FF-Replan: a baseline for probabilistic planning', *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, ICAPS 2007, Providence, Rhode Island, USA, 22–26 September, pp.352.