Robust Execution of BDI Agent Programs by Exploiting Synergies Between Intentions

Yuan Yao
School of Computer Science
University of Nottingham
Nottingham, UK
yvy@cs.nott.ac.uk

Brian Logan
School of Computer Science
University of Nottingham
Nottingham, UK
bsl@cs.nott.ac.uk

John Thangarajah
School of Computer Science and IT
RMIT University
Melbourne, Australia
john.thangarajah@rmit.edu.au

Abstract

A key advantage the reactive planning approach adopted by BDI-based agents is the ability to recover from plan execution failures, and almost all BDI agent programming languages and platforms provide some form of failure handling mechanism. In general, these consist of simply choosing an alternative plan for the failed subgoal (e.g., JACK, Jadex). In this paper, we propose an alternative approach to recovering from execution failures that relies on exploiting *positive interactions* between an agent's intentions. A positive interaction occurs when the execution of an action in one intention assists the execution of actions in other intentions (e.g., by (re)establishing their preconditions). We have implemented our approach in a scheduling algorithm for BDI agents which we call S_P. The results of a preliminary empirical evaluation of S_P suggest our approach out-performs existing failure handling mechanisms used by state-of-the-art BDI languages. Moreover, the computational overhead of S_P is modest.

1 Introduction

Arguably the dominant paradigm in agent development is the *Belief-Desire-Intention* (BDI) model (Rao and Georgeff 1991). In BDI-based agent programming languages, e.g., JACK (Winikoff 2005) *Jason* (Bordini, Hübner, and Wooldridge 2007), 2APL (Dastani 2008), the behaviour of an agent is specified in terms of beliefs, goals, and plans. *Beliefs* represent the agent's information about the environment (and itself). *Goals* represent desired states of the environment the agent is trying to bring about. *Plans* are the means by which the agent can modify the environment in order to achieve its goals. Plans are composed of *steps* which are either basic actions that directly change the agent's environment, or subgoals which are in turn achieved by other plans. For each top-level goal, the agent selects a plan which forms the root of an *intention*, and commences executing the steps in the plan. If the next step in an intention is a subgoal, a (sub)plan is selected to achieve the subgoal and pushed onto the intention, and the steps in the (sub)plan are then executed and so on. This process of repeatedly choosing and executing plans is referred to the agent's *deliberation cycle*.

Deferring the selection of plans until the corresponding goal must be achieved allows BDI agents to respond flexibly to the current state of the environment, by adapting the means used to achieve a goal to the current circumstances. However, as with any software system, BDI agent programs may fail. Several different types of execution failure can be distinguished.[1] *Coverage failures* occur when there is no applicable plan for the current (sub)goal in the current environment. *Precondition failures* occur when an action cannot be executed because a precondition for the action does not hold (e.g., in order to pick up an object, a robot agent must be within gripping distance of the object). *Postcondition failures* occur when the intended effect of the action does not hold after executing the action (e.g., if the gripper is faulty, a *pickup* action may not result in the agent holding the object).

In popular BDI agent programming languages such as JACK (Winikoff 2005) or Jadex (Pokahr, Braubach, and Lamersdorf 2005), when an execution failure occurs another applicable plan is tried to achieve the current (sub)goal. (This form of failure handling is also implemented in the *Jason* 'backtracking declarative goal' pattern (Bordini, Hübner, and Wooldridge 2007).) If no alternative means is available, then the goal is deemed failed, and the failure is propagated to higher-level motivating goals. This approach to failure handling gives rise to a very large behaviour space for BDI agent programs (Winikoff and Cranefield 2014), and can result in robust execution. However, this robustness can come at a considerable cost in terms of backtracking (i.e., partial execution of a plan to the point of failure, followed by selection of an alternative plan for the current subgoal).

In this paper, we propose an alternative approach to recovering from execution failures that relies on exploiting *positive interactions* between an agent's intentions. A positive interaction occurs when the execution of an action in one intention assists the execution of plans or actions in other intentions (e.g., by (re)establishing their context or preconditions). We have implemented our approach in a scheduling algorithm for BDI agents which we call S_P. We present the results of a preliminary empirical evaluation of S_P in a range of scenarios of increasing difficulty. The results suggest our approach out-performs existing failure handling

[1]We ignore failures arising from programmer error, and assume that programs are correctly written.

mechanisms used by state-of-the-art BDI languages. Moreover, the computational overhead of S_P is modest.

2 Beliefs, Plans and Actions

We consider BDI agent programming languages in which there is an explicit representation of beliefs, plans and actions. For simplicity, we assume that the agent's beliefs are represented by a set of literals. (In practice, beliefs in BDI agent programming languages are first order ground terms, so this assumption is really without loss of generality.) To select a plan or execute an action, a subset of these literals must hold in the current environment. In the case of a plan, the set of literals is termed the *context condition* of the plan, and indicate the situations in which the plan is applicable. In the case of an action, the literals are termed the *precondition* of the action, and indicate the situations in which the action may be executed. For example, a plan to perform a rock experiment may have the context condition 'have-battery-power'. The *postcondition* of an action is the set of literals that are made true by executing the action. A plan consists of its context condition and a sequence of steps which are either actions or subgoals. The actions in a plan are defined by their pre- and postconditions and can be either infallible or fallible.

We say an action is *infallible* if it has only one possible outcome (postcondition). An action is *fallible* if it has more than one possible outcome. We denote the effects of a fallible action by $[C_0, \ldots, C_n]$, where each C_i is a possible postcondition. Some outcomes may be more likely than others. If the probability distribution over possible outcomes is known, we denote the possible outcomes by $[(C_0, P_0), \ldots, (C_n, P_n)]$ where C_i is a possible outcome and P_i is the corresponding probability of its occurring, e.g., $[(C_0, 0.25), (C_1, 0.75)]$. For each fallible action, we assume there is a single *intended* outcome that the agent developer assumes (or hopes) will occur when the action is executed. All other (unintended) outcomes correspond to anticipated failure modes of the action. For example, the intended outcome of a fallible action may establish a precondition of a subsequent action in the plan, while the other outcome(s) do not.

We distinguish two kinds of fallible action failures: *non-disruptive* and *disruptive*. The failure of a fallible action is *non-disruptive* if it has two outcomes and the unintended outcome is the negation of the intended outcome. For example, a *pickup* action fails non-disruptively if it has $holding\_block$ and $\neg holding\_block$ as outcomes. An action which fails non-disruptively does not change the state of the environment. The agent was not holding the block before attempting the action and is not holding the block after executing the action.[2] An action failure is *disruptive* if at least one of the unintended outcomes is not the negation of the intended outcome. For example, a *pickup* action fails disruptively if it has two outcomes: $holding\_block$ and $gripper\_broken$. In addition to failing to establish the precondition of a subsequent action in the same intention, a disruptive failure may destroy the precondition of the action itself (e.g., $\neg gripper\_broken$), with the result that the action can't simply be retried.

[2]We assume that $\neg holding\_block$ is a precondition of the action, i.e., the agent would not attempt to pickup the block if it is already holding it.

3 Exploiting Synergies

An execution failure occurs when it is not possible to progress the intention chosen for execution at this cycle.[3] The 'standard' approach to execution failures in BDI agent programming is to abort the plan in which the failure occurred (perhaps after performing some 'cleanup' actions), and try another applicable plan to achieve the current subgoal. If there are no applicable plans, failure is propagated to higher-level motivating goals.

However in some situations, it may be possible to opportunistically recover from an execution failure by appropriate scheduling the agent's remaining progressable intentions to execute an already intended action which reestablishes a missing context or precondition. Actions which are executed in the same environment often have similar or overlapping preconditions; many physical actions rely on the location of the agent, for example. Alternatively, it may be possible to recover by selecting a plan for a subgoal in a progressable intention, which, in addition to achieving its triggering goal, reestablishes the missing condition. Although such opportunistic recovery may delay the achievement of the top-level goal of the failed intention (until the relevant condition can be reestablished by another intention), exploiting positive interactions between intentions can reduce the total number of actions an agent has to perform, by reducing backtracking.

We have therefore developed an approach to failure recovery in BDI agent programs, which, before backtracking is initiated, first attempts to progress the agent's non-failed intentions in such a way as to reestablish a missing precondition of an action or establish a context condition of a plan for a subgoal with no applicable plans. Our approach consists of two parts: a representation of the agent's intentions, and a scheduler, S_P, which determines, at each deliberation cycle, which intention should be progressed.

3.1 Goal-Plan Trees

We use the notion of *goal-plan trees* (Thangarajah, Padgham, and Winikoff 2003; Thangarajah and Padgham 2011) to represent the relations between goals, plans and actions, and to reason about the interactions between intentions. The root of a goal-plan tree is a top-level goal (goal-node), and its children are the plans that can be used to achieve the goal (plan-nodes). Plans may in turn contain subgoals (goal nodes), giving rise to a tree structure representing all possible ways an agent can achieve the top-level goal. In (Thangarajah, Padgham, and Winikoff 2003; Thangarajah and Padgham 2011) goal-plan trees contain only goals and plans. We extend their definition of goal-plan trees to allow actions in plans. We assume that it is possible

[3]In BDI agent platforms, postcondition failures are typically signalled by the underlying implementation of a basic action.

to generate a goal-plan tree corresponding to each top-level goal that can be achieved by an agent program.[4]

We use goal-plan trees to represent the agent's intentions. Each path through a goal plan tree corresponds to a different way of achieving the top level goal forming the root of the tree. We define the *next-step pointer* of a goal-plan tree as the next step in its corresponding intention. An intention is *progressable* if its next-step pointer points to a step which is either an action whose precondition holds, or a sub-goal for which there is at least one applicable plan in the current state. The execution of an agent program thus corresponds to an interleaving of a path through each of the goal-plan trees corresponding to the agent's current intentions.

3.2 The S_P Scheduler

The S_P scheduler is based on Monte-Carlo Tree Search (MCTS) (Chaslot et al. 2006; Kocsis and Szepesvári 2006; Chaslot et al. 2008). MCTS is a best-first search in which pseudorandom simulations are used to guide expansion of the search tree. It was originally developed for game playing (e.g., Go (Chaslot et al. 2006)). However, it has also been applied in problems where the domain can be represented as trees of sequential decisions (e.g. planning).

S_P schedules the execution of agent's intentions, i.e., it chooses which intention should be progressed at the current deliberation cycle. The scheduler takes three parameters as input: the set of goal plan trees (and their next-step pointers) corresponding to the agent's current intentions, T, the current state of the agent's environment s_0, and the number of iterations to be performed α. α specifies the 'computational budget' of the scheduler, and allows the agent developer to configure how long the scheduler should run in a particular application. The pseudocode for the scheduler is shown in Algorithm 1.

Algorithm 1 Return the action be executed at this cycle

```
function $S_P(T, s_0, \alpha)$
    $n_0 \leftarrow node_0(T, s_0)$
    for $i \leftarrow 1, \alpha$ do
        $n_e \leftarrow$ MAX-UCB1-LEAF-NODE$(n_0)$
        $children(n_e) \leftarrow$ EXPAND$(n_e)$
        $n_s \leftarrow$ RANDOM-CHILD$(children(n_e))$
        $value(n_s) \leftarrow$ SIMULATE$(n_s)$
        BACKUP$(value(n_s), n_s)$
    return BEST-CHILD$(n_0)$
```

As in MCTS, S_P iteratively builds a search tree. Each node in the search tree represents an interleaving of steps from the goal-plan trees in T, and records the state of the agent's environment resulting from the execution of this interleaving and the current next-step in each goal-plan tree. In addition, each node n also contains values $num(n)$ representing the number of times it has been visited and $value(n)$ representing the total reward value of all simulations starting from the state represented by this node or any of its children. Edges in the search tree represent the selection of a plan for a subgoal or the execution of primitive action in a plan.

[4]Note that goal-plan trees can be computed offline.

Each iteration of the main loop consists of 4 phases: selection, expansion, simulation and back-propagation. In the *selection* phase, a leaf node n_e is selected for expansion. A node may be expanded if it represents a non-terminal state (a state in which it is possible to execute the next-step of a goal-plan tree in T). The node is selected using Upper Confidence Bound for Trees (UCB1) (Auer, Cesa-Bianchi, and Fischer 2002), which models the choice of node as a k-armed bandit problem (Kocsis and Szepesvári 2006; Auer, Cesa-Bianchi, and Fischer 2002). Starting from the root node, we recursively follow child nodes with highest UCB1 value until a leaf node is reached.

In the *expansion* phase, n_e is expanded by adding child nodes representing the execution of the next-step of each intention that is progressable in the environment state $s(n_e)$. If the next-step is a subgoal, then a child node n' is generated for each plan for the subgoal that is applicable in $s(n_e)$ (where $s(n') = s(n_e)$), and the next-step pointer of n' points to the first action in the applicable plan. If the next-step is an infallible action, a single child node is generated which represents the state resulting from the execution of the action and the advancement of the next-step pointer to the next action or subgoal in the plan. The expansion of a fallible action generates multiple child nodes, one for each possible outcome of the action. Each possible outcome of a fallible action is thus considered separately, as "opponent moves" in a two player game. Each child node therefore corresponds to a different choice of which intention to progress at this cycle, and the different postconditions the chosen step may bring about. One of the newly created child nodes, n_s, is then selected at random for simulation.

In the *simulation* phase, the reward value of n_s is estimated. Starting from the state represented by n_s, a next-step of a goal-plan tree that is executable in that state is randomly selected and executed, and the environment and the current step of the selected goal-plan tree updated. This process is repeated until a terminal state is reached in which no next-steps can be executed or all top-level goals are achieved. The reward value $value(n_s)$ is then taken to be the number of top-level goals achieved in the terminal state.

Finally, in the *back-propagation* phase $value(n_s)$ is back-propagated from n_s to all nodes n on the path from n_s to the root node n_0. For each node n, the value of $num(n)$ is increased by 1, and $value(n)$ is increased by $value(n_s)$.

After α iterations, the algorithm halts, and an execution step corresponding to the most visited child of the root node (the child with highest $num(n)$ value) is returned (this selection strategy is called *robust child* in (Schadd 2009)). S_P thus tends to select the most "promising" step at each deliberation cycle, by simulating possible executions of agent's current intentions.

3.3 Robust Execution

In the selection phase, only progressable intentions can be selected for expansion. When an execution failure occurs, i.e., the precondition of the next-step of an intention $t_i \in T$ is false in the current environment state, s_0, S_P does not backtrack to try an alternative plan for the current subgoal in t_i immediately as in JACK or Jadex. Instead S_P attempts to find an execution order of the steps in progressable intentions which re-establishes the context or pre- condition of the next step in t_i. Only when all the agent's remaining intentions become non-progressable, does S_P drop the currently intended plan for the current subgoal in each intention, and consider an alternative applicable plan for each subgoal. If all applicable plans for a subgoal have been tried and failed, then the current subgoal is dropped and failure is propagated to higher-level goals.

4 Evaluation

In this section, we evaluate the ability of our approach to recover from execution failures. We compare the number of goals achieved and the number of 'backtracks' (selection of an alternative plan for a subgoal) for both S_P and Round Robin scheduling with failure handling ($RR+$) in scenarios of increasing difficulty. In $RR+$, intentions are executed in RR fashion. When an execution failure occurs, the current intended plan is dropped and an alternative applicable plan is tried. If there are no applicable plans, the current subgoal is dropped and failure is propagated to higher-level goals. The number of goals achieved is a key metric for any BDI agent. The number of 'backtracks' can be seen as the amount of effort 'wasted' by the agent in achieving its goals (i.e., plans that are partially executed before being dropped when they become unexecutable in favour of an alternative plan for a subgoal). We chose $RR+$ as representative of the state of the art in practical implementations of agent programming languages such as JACK and Jadex.

4.1 Experimental Setup

In the interests of generality, our evaluation is based on sets of randomly-generated, synthetic goal-plan trees representing the current intentions of an agent in a simple static environment. By controlling the characteristics of the trees, and the number of fallible actions, we can evaluate the performance of each approach under different conditions.

The environment is defined by a set of propositions, V, that may appear as pre- or postconditions of the actions in each goal-plan tree. By varying the number of propositions in V, we can vary the likelihood of actions in different goal-plan trees having the same pre- and postconditions, and hence the probability of positive and negative interactions between intentions. Each synthetic goal-plan tree is specified by six parameters: the depth of the tree, the plan branching factor (the maximum number of plans that can be used to achieve a goal), the goal branching factor (the maximum number of sub-goals a plan may have), the maximum number of actions in a plan, the probability that an action is a fallible action, and the number of environment variables that may appear in the tree. For top-level goals where there are two plans to achieve the goal, if one of the plans has the environment variable p as its precondition, then the other plan has $\neg p$ as its precondition. For subplans, the context condition of the plan is established by the postcondition of a previous step in the goal-plan tree. Each plan consists of a list of actions followed by subgoals. The first action in the plan has the plan's context condition as its own precondition. The remaining actions either have the plan's context condition as their own precondition or their precondition is established by a previous action in the plan. The postcondition of an action is selected randomly from the set of environment variables V. Execution of plans and actions in different trees therefore may interact through shared environment variables, and a change in the value of a variable may affect (positively or negatively) the selection of plans or the executability of actions in more than one tree. If all actions are infallible (i.e., the percentage of fallible actions is 0), the constraints on goal-plan tree structure and preconditions ensure that: (a) each plan is well formed (the plan can be successfully executed in some environment), and (b) taken individually, each goal-plan tree is executable.

Each trial consists of a randomly generated environment and a randomly generated set of 10 goal-plan trees (each generated with the same parameter values). The variables appearing as conditions in each tree are randomly selected from the environment variables, subject to the constraints specified above. The depth of each tree is 5, and we assume there are exactly two plans for each goal and that each plan has a single environment variable as its context condition. Each plan consists of three actions followed by one subgoal (except for leaf plans, which have no subgoals).

For the experiments reported below, we vary the number of environment variables appearing in pre- and postconditions from 20 to 50 (with fewer variables, there are more interactions between the goal-plan trees). The percentage of fallible actions in each plan was varied from 0% (all actions are infallible actions) to 30%, and in a given experiment, all fallible actions in a goal-plan tree either fail non-disruptively or fail disruptively. We also varied the probability of action failure (i.e., probability of the non-intended postcondition of a fallible action occurring) from 10% to 50%. For each experiment we report average performance values over 50 trials. S_P was configured to perform 1000 iterations ($\alpha = 1000$).

4.2 Results

We evaluate the performance of each approach in a range of scenarios. In our first scenario, all fallible actions fail non-disruptively. When the goal-plan trees are generated, each fallible action added to a plan is assigned two postconditions, p and $\neg p$, where p is randomly selected from V.

The results of the first experiment are shown in Tables 1 and 2. In the tables, $|V|$ is the number of environment variables and ND is the percentage of fallible actions in each plan (0%, 5%, 10%, 20% and 30%). *Goal* is the average number of top-level goals achieved by each approach, and *Back* is the average number of backtracks. For each percentage of fallible actions considered, the first row gives results for an action failure probability of 10%, and the second and third rows give results for action failure probabilities of 25% and 50%.

As can be seen, in cases where there is little interaction between the goal plan trees ($|V| = 50$), and the percentage of fallible actions in plans is small, the average number of goals achieved by $RR+$ and S_P is similar (9.32 for $RR+$ vs 10 for S_P). However even in this setting, $RR+$ requires significantly more backtracking to achieve the goals (13.82 vs

Table 1: $RR+$; all fallible actions fail non-disruptively

	$\lvert V\rvert = 50$		$\lvert V\rvert = 40$		$\lvert V\rvert = 30$		$\lvert V\rvert = 20$	
ND	Goal	Back	Goal	Back	Goal	Back	Goal	Back
0%	9.32	13.82	9.34	15.82	9.32	21.50	8.26	30.26
	9.32	14.16	9.32	16.36	9.22	20.92	8.22	30.96
5%	9.30	14.48	9.26	17.88	9	23.06	8.20	30.26
	9.18	16.58	8.92	18.74	8.88	23.50	8	32.46
	9.26	14.88	9.14	17.34	9.12	22.92	8.20	29.32
10%	9.16	16.06	9.14	18.82	8.72	25.06	8.08	29.68
	8.88	19.90	8.70	23.22	8.46	25.70	7.90	33.94
	9.22	16	9.10	19.02	9.06	21.88	8.08	30.82
20%	8.98	19.26	8.94	22.26	8.48	25.60	7.72	32.32
	8.38	25.78	8.30	26.58	7.90	29.64	7.18	39.2
	9.04	18.18	9.08	19.18	8.94	24.90	7.94	33.46
30%	8.78	21.74	8.34	25.02	8.12	28.56	7.64	37.14
	7.62	30.96	7.24	34.60	7.42	37.18	6.56	44.78

Table 2: S_P; all fallible actions fail non-disruptively

	$\lvert V\rvert = 50$		$\lvert V\rvert = 40$		$\lvert V\rvert = 30$		$\lvert V\rvert = 20$	
ND	Goal	Back	Goal	Back	Goal	Back	Goal	Back
0%	10	0	10	0	10	0	10	0
		0		0		0.04		0.02
5%	10	0.02	10	0.02	10	0.04	10	0.04
		0		0.16		0.04		0.04
		0.02		0.04		0		0
10%	10	0.04	10	0.14	10	0.06	10	0
		0.24		0.12		0.50		0.22
		0.16		0.08		0.20		0.20
20%	10	0.38	10	0.36	10	0.24	10	0.22
		1.78		1.02		0.82		0.64
		0.52		0.48		0.48		0.28
30%	10	1.48	10	0.98	10	1.10	10	0.78
		6.16		3.58		2.26		1.76

Table 3: $RR+$; all fallible actions fail disruptively

	$\lvert V\rvert = 50$		$\lvert V\rvert = 40$		$\lvert V\rvert = 30$		$\lvert V\rvert = 20$	
ND	Goal	Back	Goal	Back	Goal	Back	Goal	Back
0%	9.32	13.82	9.34	15.82	9.32	21.50	8.26	30.26
	9.32	13.38	9.32	17.24	9.04	21.22	8.22	28.28
5%	9.16	14.36	9.06	16.58	9.02	22.16	8.2	28.66
	9.08	15.02	9	17.58	9.04	22.04	8.2	28.08
	9.30	13.68	9.26	16.54	8.92	21.54	8.12	27.78
10%	9.18	14.74	8.98	18.22	8.8	21.5	8.04	28.62
	9.04	15.76	8.94	19.34	8.76	24.08	7.78	30.36
	9.22	14.24	9.20	17.56	8.80	22.04	8.10	29.30
20%	9.10	16.20	9.02	19.88	8.70	24.10	7.88	30.04
	9	19.76	8.64	21.56	8.52	26.24	7.72	35.84
	9.20	14.44	9.16	17.42	8.68	20.72	7.88	30.96
30%	9.10	17.16	9	20.86	8.44	23.48	7.82	30.84
	8.68	20.62	8.52	34.6	8.14	31.44	7.52	36.92

Table 4: S_P; all fallible actions fail disruptively

	$\lvert V\rvert = 50$		$\lvert V\rvert = 40$		$\lvert V\rvert = 30$		$\lvert V\rvert = 20$	
ND	Goal	Back	Goal	Back	Goal	Back	Goal	Back
0%	10	0	10	0	10	0	10	0
		0		0		0.02		0
5%	10	0	10	0.02	10	0.02	10	0.02
		0.04		0		0		0
		0.02		0		0		0.02
10%	10	0.02	10	0	10	0.02	10	0.04
		0.06		0.02		0.04		0
		0.02		0		0.06		0.02
20%	10	0.14	10	0.10	10	0.24	10	0.02
		0.32		0.12		0.12		0.20
		0.18		0.10		0.22		0.18
30%	10	0.22	10	0.20	10	0.26	10	0.20
		1.08		0.76		1.14		0.46

0). Although the number of interactions between intensions is small, actions in one intention may destroy the context conditions of plans or the preconditions of actions in other intentions. In contrast S_P is able to schedule the execution of intentions so as to avoid negative interactions, achieving all 10 goals, and in general no backtracking is required.

As the percentage of fallible actions in plans, and the probability of each fallible action failing increases, the average number of goals achieved by $RR+$ decreases, and the number of backtracks required to achieve these goals increases. When 30% of actions are fallible, and each action has a 50% chance of failure, $RR+$ can achieve only 7.62 goals on average, and performs on average 30.96 backtracks to do so. The decrease in performance of $RR+$ relative to the case in which all actions are deterministic is due to the difficulty of recovering from action failures using only backtracking. S_P achieves all 10 goals and requires only 6.16 backtracks on average, demonstrating the utility of exploiting positive interactions.

As the number of environment variables decreases and the degree of interaction between the agent's intentions increases, the performance of $RR+$ declines, even when all actions in plans are deterministic. In the $|V| = 20$, $RR+$ achieves 8.26 goals on average and requires 30.26 backtracks, which is only a marginal improvement on the low interaction/high failure case. With high interaction and high failure of actions, the performance of $RR+$ declines significantly to 6.56 goals and 44.79 backtracks. In contrast, even in the extremely challenging setting, S_P achieves all 10 goals, and requires only 1.76 backtracks on average. The number of backtracks is lower than in the low interaction ($|V| = 50$) case, as S_P is able to exploit the larger number of interactions between intentions to recover from failures without backtracking in this case.

In our second scenario, all fallible actions fail disruptively. Each fallible action has two possible outcomes, $p, q \in V$, where $q \neq \neg p$. The results of the second experiment are shown in Tables 3 and 4. The presentation of results is as in the first experiment. As can be seen, in general, disruptive actions facilitate failure recovery, as there is a higher probability that the unintended outcome of a fallible action will reestablish a condition necessary to progress a failed intention. Rather than being restricted to the negation of the intended outcome of an action, in this setting the unintended outcome of an action may be any environment variable in V. $RR+$ achieves more goals and requires fewer backtracks in the low interaction / high failure case, 8.68 goals and 20.62 backtracks compared to 7.62 goals and 30.96 backtracks when actions fail non-disruptively. This increase in performance also extends to the high interaction / high failure case, where $RR+$ achieves 7.52 goals and requires 36.92 backtracks compared to 6.56 goals and 44.78 backtracks in the first experiment. As in the first experiment, S_P achieves 10 goals in all cases, but the number of backtracks required is significantly reduced. S_P requires 1.08 backtracks on average to achieve 10 goals when 30% of actions in plans are fallible, and actions fail with 50% probability in the low interaction ($|V| = 50$) case, and only 0.46 backtracks in the high interaction ($|V| = 20$) case.

Overall, S_P is able to avoid negative interactions between intentions, while exploiting positive interactions to recover from execution failures with minimal backtracking. It might be argued that the degree of interactions between intentions in our experiments is higher than typically occurs in agent programs. For example, if the agent has only a single intention and it fails, S_P cannot use actions in other intentions to recover from the failure. To determine the extent to which the performance of S_P relies of the agent executing a 'large' number of intentions in parallel, we repeated the experiments above with five goal-plan trees in each trial (i.e., the agent executes five intentions in parallel). Even in this setting, S_P significantly out-performs $RR+$. For example, in the best case for $RR+$ ($|V| = 50$, plans contain 5% disruptive actions with probability of failure = 10%) $RR+$ achieves 4.82 goals with 3.8 backtracks, while S_P achieves 5 goals without backtracking. In the worst case for $RR+$ ($|V| = 20$, plans contain 30% actions that fail non-disruptively with probability 50%) $RR+$ achieves 3.76 goals with 15.86 backtracks while S_P achieves 5 goals with 2.38 backtracks. Full results are omitted due to lack of space.

4.3 Computational Overhead

The computational overhead of S_P depends on the search configuration α: i.e., how many iterations of the algorithm are performed. With the search configuration used for the experiments above ($\alpha = 1000$), S_P requires 35 milliseconds to return the action to be executed at this deliberation cycle. If actions require significant time to execute, e.g., moving from one location to another, this is a relatively small overhead, particularly when set against the time spent partially executing plans which ultimately fail in the $RR+$ approach.

As S_P is an anytime algorithm, the time required to return the next action can be reduced by reducing the number of iterations performed by the algorithm. Reducing α has a relatively small impact on the ability of S_P to recover from execution failures. For example, with $\alpha = 100$, S_P achieves 9.98 goals with 4.18 backtracks in the case where $|V| = 20$, plans contain 30% non-disruptive actions with probability of failure = 50% (cf Table 2), and requires 3ms to return the action to be executed at the current cycle.

5 Related Work

Flexible and *robust* execution in dynamic environments are key characteristics of BDI agent systems. Flexibility arises from having multiple ways of achieving tasks, and robustness from how the system recovers from failure. However, AgentSpeak(L) (Rao 1996), the most influential abstract BDI agent programming language, does not incorporate mechanisms for dealing with plan failure. The Conceptual Agent Notation (CAN) (Winikoff et al. 2002) improves on AgentSpeak(L) by providing both declarative and procedural notions of goals and a mechanism for retrying goal achievement via alternative plans if one plan fails. This 'retry on failure' mechanism is built into practical agent programming languages such as JACK (Winikoff 2005) and Jadex (Pokahr, Braubach, and Lamersdorf 2005). These languages also allow the programmer to define 'clean up' actions that are executed when a plan fails. In Jason (Bordini, Hübner, and Wooldridge 2007), a plan failure triggers a goal deletion event that may also initiate clean up actions including attempting alternative plans to achieve the goal. 3APL (Dastani, van Riemsdijk, and Meyer 2005) provides greater support for recovery, by allowing a replanning mechanism that allows a failed plan to be either revised allowing execution to continue, or dropped with possible clean up actions. The Cypress architecture (Wilkins et al. 1995) combines the the Procedural Reasoning System reactive executor PRS-CL and the SIPE-2 look-ahead planner. PRS-CL is used to pursue agent's intentions using a library of plans. If a failure occurs, the executor calls SIPE-2 to produce a new plan. However, this approach focusses on the generation of new plans to recover from failures, rather than interleaving current intentions. In (Yao, Logan, and Thangarajah 2014), a variant of MCTS called Single-Player Monte-Carlo Tree Search (Schadd et al. 2012) is used to schedule intentions at the level of plans rather than individual actions. There has also been work on avoiding conflicts and redundant actions in a multi-agent setting. For example, in (Ephrati and Rosenschein 1993) an approach to planning and interleaving the execution of tasks by multiple agents is presented, where the combined execution agent tasks achieves a global goal. They show how conflicts between intentions and redundant steps can be avoided by appropriate scheduling of the actions of the agents. However none of these approaches consider exploiting synergies to recover from execution failures as we propose here.

6 Discussion and Future Work

We presented an approach to recovering from execution failures in BDI agent programs which exploits *positive interactions* between an agent's intentions to reestablish context or preconditions. The results of a preliminary empirical evaluation of S_P, a scheduler based on our approach, suggest it out-performs failure handling mechanisms used by state-of-the-art BDI languages. S_P's performance advantage is greatest in those scenarios that are most challenging for conventional approaches (high interaction / high failure), as it is able to exploit positive interactions to reduce backtracking while at the same time avoiding negative interactions between intentions. For simplicity, our evaluation of S_P focusses on a static environment. However we stress that S_P is not limited to static environments. The choice of action at each deliberation cycle is based on the current state of the agent's environment at that cycle, coupled with simulations of the possible outcomes of actions.

In future work, we plan to extend S_P to exploit positive interactions to reduce the number of executed steps when intentions do not fail. We also plan to investigate the incorporation of simple environment models to allow the prediction of likely environment changes during simulation.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Bordini, R. H.; Hübner, J. F.; and Wooldridge, M. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley.

Chaslot, G.; Saito, J.; Bouzy, B.; Uiterwijk, J. W. H. M.; and van den Herik, H. J. 2006. Monte-Carlo strategies for computer Go. In *Proceedings of the 18th Belgian-Dutch Conference on Artificial Intelligence*, 83–90.

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-Carlo tree search: A new framework for game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Dastani, M.; van Riemsdijk, M. B.; and Meyer, J.-J. C. 2005. Programming multi-agent systems in 3APL. In Bordini, R. H.; Dastani, M.; Dix, J.; and Fallah-Seghrouchni, A. E., eds., *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer. 39–67.

Dastani, M. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16(3):214–248.

Ephrati, E., and Rosenschein, J. S. 1993. A framework for the interleaving of execution and planning for dynamic tasks by multiple agents. In Castelfranchi, C., and Müller, J., eds., *From Reaction to Cognition, 5th European Workshop on Modelling Autonomous Agents, MAAMAW '93, Neuchatel, Switzerland, August 25-27, 1993, Selected Papers*, 139–153. Springer.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *17th European Conference on Machine Learning*, 282–293. Berlin, Germany: Springer.

Pokahr, A.; Braubach, L.; and Lamersdorf, W. 2005. Jadex: A BDI reasoning engine. In Bordini, R.; Dastani, M.; Dix, J.; and El Fallah Seghrouchni, A., eds., *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer US. 149–174.

Rao, A. S., and Georgeff, M. P. 1991. Modeling rational agents within a BDI-architecture. In Allen, J. F.; Fikes, R.; and Sandewall, E., eds., *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, 473–484. Cambridge, MA, USA: Morgan Kaufmann.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World: Agents Breaking Away*, MAAMAW '96, 42–55. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Schadd, M. P. D.; Winands, M. H. M.; Tak, M. J. W.; and Uiterwijk, J. W. H. M. 2012. Single-player Monte-Carlo tree search for SameGame. *Knowledge-Based Systems* 34:3–11.

Schadd, F. C. 2009. Monte-Carlo search techniques in the modern board game Thurn and Taxis. Master's thesis, Maastricht University.

Thangarajah, J., and Padgham, L. 2011. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning* 47(1):17–56.

Thangarajah, J.; Padgham, L.; and Winikoff, M. 2003. Detecting & avoiding interference between goals in intelligent agents. In Gottlob, G., and Walsh, T., eds., *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, 721–726. Acapulco, Mexico: Morgan Kaufmann.

Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1):121–152.

Winikoff, M., and Cranefield, S. 2014. On the testability of BDI agent systems. *Journal of Artificial Intelligence Research* 51:71–131.

Winikoff, M.; Padgham, L.; Harland, J.; and Thangarajah, J. 2002. Declarative & procedural goals in intelligent agent systems. In Fensel, D.; Giunchiglia, F.; McGuinness, D. L.; and Williams, M.-A., eds., *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, 470–481. Toulouse, France: Morgan Kaufmann.

Winikoff, M. 2005. JACK intelligent agents: An industrial strength platform. In Bordini, R. H.; Dastani, M.; Dix, J.; and Fallah-Seghrouchni, A. E., eds., *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer. 175–193.

Yao, Y.; Logan, B.; and Thangarajah, J. 2014. SP-MCTS-based intention scheduling for BDI agents. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI-2014)*, 1133–1134. Prague, Czech Republic: ECCAI.