On the testability of BDI agents*

Michael Winikoff and Stephen Cranefield

Department of Information Science, University of Otago,
P.O. Box 56, Dunedin, New Zealand
{michael.winikoff,stephen.cranefield}@otago.ac.nz

Abstract. Before deploying a software system we need to assure ourselves that the system will behave correctly. This assurance is usually done by testing the system. However, it is intuitively obvious that adaptive systems, including agent-based systems, can exhibit complex behaviour, and are thus harder to test. In this paper we examine this intuition in the case of Belief-Desire-Intention (BDI) agents. We analyse the size of the behaviour space of BDI agents and show that although the intuition is correct, we found that the introduction of failure handling had a much larger effect on the size of the behaviour space than we expected. We also discuss the implications of these findings on the testability of BDI agents.

1 Introduction

As agent-based systems are increasingly deployed, the issue of *assurance* rears its head. Before deploying a system, we need to convince those who will rely on the system (or those who will be responsible if it fails) that the system will, in fact, work. Traditionally, this assurance is done through testing. However, it is generally accepted that adaptive systems exhibit a wider and more complex range of behaviours, making testing harder. For example, Munroe et al. made the following observation: "*[V]alidation through extensive tests was mandatory .... However, the task proved challenging for several reasons. First, agent-based systems explore realms of behaviour outside people's expectations and often yield surprises ...*" [1, Section 3.7.2].

That is, there is an intuition that agent systems exhibit complex behaviour, which makes them harder to test. In this paper we explore this intuition, focusing on the well known Belief-Desire-Intention (BDI) [2, 3] approach to realising adaptive and flexible agents, which has been demonstrated to be practically applicable, resulting in reduced development cost and increased flexibility [4].

We explore the intuition that "agent systems are harder to test" by analysing both the space of possible behaviours of BDI agents, and the probability of failure. We focus on BDI agents both because they provide a well-defined execution mechanism that can be analysed, but also because we seek to understand the complexities (and testability implications) of adaptive and intelligent behaviour in the absence of parallelism (since the implications of parallelism are already known [5, Section 3]).

Specifically, we aim to answer these questions:

1. How large is the behaviour space for BDI agents?

* This paper is available online at http://eprints.otago.ac.nz/1015/

2. What factors influence the size of the behaviour space?
3. Is it feasible to assure the effectiveness of BDI systems by testing?

As might be expected, we show that the intuition that "agent systems are harder to test" is correct, i.e. that agent systems have many traces. The contribution of this paper is to confirm the intuition by *quantifying* the size of the behaviour space. We also find some surprising results about what factors influence the size of the behaviour space.

Although there has recently been increasing interest in testing agent systems [6–9], there has been surprisingly little work on determining the feasibility of testing agent systems in the first place. Padgham and Winikoff [10, pages 17-19] analyse the number of successful executions of a BDI agent's goal-plan tree (defined in Section 3), but they do not consider failure or failure handling in their analysis, nor do they consider testability implications. Shaw et al. [11] have analysed goal-plan trees and shown that checking whether a goal-plan tree has an execution schedule with respect to resource requirements is NP-complete. This is a different problem to the one that we tackle: they are concerned with the allocation of resources amongst goals, rather than with the behaviour space.

The remainder of the paper is structured as follows. We begin by briefly presenting the BDI execution model (Section 2) and discussing how BDI execution can be viewed as a process of transforming goal-plan trees (Section 3). Section 4 is the core of the paper where we analyse the behaviour space of BDI agents. We then consider how our analysis and its assumptions hold up against a real system (Section 5) before discussing the implications for testing and concluding the paper in Section 6.

2 The BDI Execution Model

In the implementation of a BDI agent the key concepts are *beliefs* (or, more generally, data), *events* and *plans*. The reader may find it surprising that goals are not key concepts in BDI systems. The reason is that goals are modelled as events: the acquisition of a new goal is viewed as a "new goal" event, and the agent responds by selecting and executing a plan that can handle that event[1]. In the remainder of this section, in keeping with established practice, we will describe BDI plans as handling events (not goals).

A BDI plan consists of three parts: an *event pattern* specifying the event(s) it is relevant for, a *context condition* (a Boolean condition) that indicates in what situations the plan can be used, and a *plan body* that is executed. A plan's event pattern and context condition may be terms containing variables, so a matching or unification process (depending on the particular BDI system) is used by BDI interpreters to find plan instances that respond to a given event. In general the plan body can contain arbitrary code in some programming language[2], however for our purposes (following abstract notations such as AgentSpeak(L) [13] and CAN [14]) we assume that a plan body is a sequence of steps, where each step is either an action[3] (which can succeed or fail) or an event

[1] Other types of event typically include the addition and removal of beliefs.

[2] For example, in JACK [12] a plan body is written in a language that is a superset of Java.

[3] This includes both traditional actions that affect the agent's environment, and internal actions that invoke code, or that check whether a certain condition follows from the agent's beliefs.

to be posted. Note that this assumption does not make our analysis specific to AgentSpeak(L): in fact AgentSpeak(L) can be seen as an abstract language that captures the essence of a range of (more complex) BDI languages.

A BDI execution cycle is an elaboration of the following event-driven process[4]:

1. An event occurs (either received from an outside source, or triggered from within the agent).
2. The agent determines a set of instances of plans in its plan library with event patterns that match the triggering event. This is the set of *relevant* plan instances.
3. The agent evaluates the context conditions of the relevant plan instances to generate the set of *applicable* plan instances. This is done by unifying each context condition in turn against the belief base, with backtracking used to consider all possible solutions. If there are no applicable plan instances then the event is deemed to have failed, and if it has been posted from a plan, then that plan fails. Note that a single relevant plan may lead to no applicable plan instances (if the context condition is false), or to more than one applicable plan instance (if the context condition, which may contain free variables, has multiple solutions).
4. One of the applicable plan instances is selected and is executed. The selection mechanism varies between platforms. For generality, our analysis does not make any assumptions about plan selection. The plan's body may create additional events that are handled using this process.
5. If the plan body fails, then failure handling is triggered.

For brevity, in the remainder of the paper we will use the term "plan" loosely to mean either a plan or plan instance where the distinction is not significant.

This execution cycle generates an execution trace comprising the sequence of actions that were attempted, together with a flag indicating their success or failure. A *failed* trace is one where there is a plan failure that is not handled using the failure handling mechanism. A *successful* trace is one where all failures are handled (i.e. recovered from) at some level.

There are a few approaches to dealing with failure. Perhaps the most common approach, which is used in many of the existing BDI platforms, is to select an alternative applicable plan, and only consider an event to have failed when there are no remaining applicable plans. In determining alternative applicable plans one may either consider the existing set of applicable plans, or recalculate the set of applicable plans (ignoring those that have already been tried). This makes sense because the situation may have changed since the applicable plans were determined. Many (but not all) BDI platforms use the same failure-handling mechanism of retrying plans upon failure, and our analysis applies to all of these platforms.

Testing is, in essence, the process of repeatedly running a system, and checking whether each observed behaviour trace is "correct" (i.e. it conforms to a specification, which we do not model). In the context of BDI agents, a successful execution may exhibit behaviour that is not correct; for instance, a traffic controller agent may successfully execute actions that set all traffic signals at an intersection to green and achieve a

[4] BDI engines are, in fact, more complicated than this as they can interleave the execution of multiple active plan instances (or *intentions*) that were triggered by different events.

goal by doing so. It is also possible for a failed execution to be correct, for instance, if a traffic controller agent is attempting to route cars from point *A* to point *B*, but a traffic accident has blocked a key bridge between these two points, then the rational (and correct) behaviour for the agent is to fail to achieve the goal.

3 BDI Execution as Goal-Plan Tree Expansion

BDI execution is a dynamic process that progressively executes actions as goals are posted. In order to more easily analyse this process we now present an alternative view that is more declarative. Instead of viewing BDI execution as a process, we view it as a data transformation, from a *goal-plan tree* into a sequence of action executions.

The events and plans can be visualised as a tree where each goal has as children the plan instances that are applicable to it, and each plan instance has as children the sub-goals that it posts. This *goal-plan tree* is an "and-or" tree: each goal is realised by one of its plan instances ("or") and each plan instance needs all of its sub-goals to be achieved ("and").

Viewing BDI execution in terms of a goal-plan tree and action sequences makes the analysis of the behaviour space size easier. We consider BDI execution as a process of taking a goal-plan tree and transforming it into a sequence recording the (failed and successful) executions of actions, by progressively making decisions about which plans to use for each goal and executing these plans.

This process is non-deterministic: we need to choose a plan for each goal in the tree. Furthermore, when we consider failure, we need to consider for each action whether it fails or not, and if it does fail, what failure recovery is done.

We now define the transformation process in detail. Prolog code implementing the process can be found in Figure 1. It defines a non-deterministic predicate `exec` with its first argument being the (input) goal-plan tree, and the second argument an (output) sequence of actions. A goal-plan tree is represented as a Prolog term conforming to the following simple grammar (where *GPT* abbreviates "Goal-Plan Tree", *AoGL* abbreviates "Action or Goal List", and *A* is a symbol):

$$\begin{aligned}
\langle GPT \rangle &::= \texttt{goal([])} \mid \texttt{goal([}\langle PlanList \rangle\texttt{])} \\
\langle PlanList \rangle &::= \langle Plan \rangle \mid \langle Plan \rangle\texttt{,}\langle PlanList \rangle \\
\langle Plan \rangle &::= \texttt{plan([])} \mid \texttt{plan([}\langle AoGL \rangle\texttt{])} \\
\langle AoGL \rangle &::= \texttt{act(}A\texttt{)} \mid \langle GPT \rangle \mid \texttt{act(}A\texttt{),}\langle AoGL \rangle \mid \langle GPT \rangle\texttt{,}\langle AoGL \rangle
\end{aligned}$$

For example, a goal with two applicable plans, each containing a single action, is represented by the term `goal([plan([act(a)]), plan([act(b)])])`.

In our analysis we make a simplifying assumption. Instead of modelling the instantiation of plans to plan instances, we assume that the goal-plan tree contains applicable plan instances. Thus, in order to transform a goal node into a sequence of actions we (non-deterministically) select one of its applicable plan instances. The selected plan is then transformed in turn, resulting in an action sequence (line 2 in Figure 1). When selecting a plan, we consider the possibility that any of the applicable plans could be

```
1  exec(goal([]),[]).
2  exec(goal(Plans), Trace) :- remove(Plans,Plan,Rest), exec(Plan,Trace1),
3      (failed(Trace1) -> recover(Rest,Trace1,Trace) ; Trace=Trace1).
4  exec(plan([]), []).
5  exec(plan([Step|Steps]), Trace) :- exec(Step,Trace1),
6      (failed(Trace1) -> Trace=Trace1 ; continue(Steps,Trace1,Trace)).
7  exec(act(Action), [Action]).
8  exec(act(Action), [Action,fail]).
9  failed(Trace) :- append(X,[fail],Trace).
10 recover(Plans,Trace1,Traces) :-
11     exec(goal(Plans),Trace2), append(Trace1,Trace2,Traces).
12 continue(Steps,Trace1,Trace) :- exec(plan(Steps),Trace2),
13     append(Trace1,Trace2,Trace).
14 % remove(A,B,C) iff removing element B from list A leaves list C
15 remove([X|Xs],X,Xs).
16 remove([X|Xs],Y,[X|Z]) :- remove(Xs,Y,Z).
```

Fig. 1. Prolog Code to Expand Goal-Plan Trees

chosen, not just the first plan. This is done because at different points in time different plan instances may be applicable.

If the selected plan executes successfully (i.e. the action trace doesn't end with a fail marker; line 9), then the resulting trace is the trace for the goal's execution (line 3). Otherwise, we perform failure recovery (line 10), which is done by taking the remaining plans (i.e. excluding the plan that has already been tried) and transforming the goal with these plans as options. The resulting action sequence is appended to the action sequence of the failed plan to obtain a complete action sequence for the goal.

This process can easily be seen to match that described in Section 2 (with the exception, discussed above, that we begin with applicable plans, not relevant plans). Specifically, an applicable plan is selected and executed, and if it is successful then execution stops. If it is not successful, then an alternative plan is selected and execution continues (i.e. action sequences are appended).

In order to transform a plan node we first transform the first step of the plan, which is either a sub-goal or an action (line 5). If this is successful, then we continue to transform the rest of the plan, and append the two resulting traces together (lines 6 and 12). If the first step of the plan is not successful, then the trace is simply the trace of the first step (line 6), in other words we stop transforming the plan when a step fails.

Finally, in order to transform an action into an action sequence we simply take the action itself as a singleton sequence (line 7). However, we need to take into account the possibility that an action may fail, and thus a second possibility is the action followed by a failure indicator (line 8). Note that in our model we don't concern ourselves with why an action fails: it could be due to lack of resources, or other environmental issues.

4 Behaviour Space Size of BDI Agents

We now consider how many possible behaviours there are for a BDI agent that is trying to realise a goal with a given goal-plan tree. We use the analysis of the previous section as our basis, that is, we view BDI execution as transforming a goal-plan tree into action traces. Thus, the question of how large is the behaviour space for BDI agents,

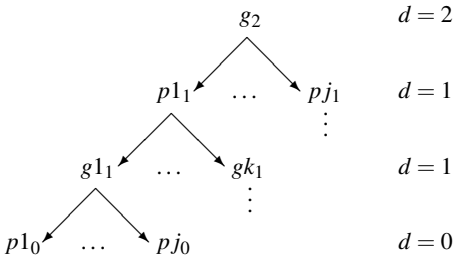

Fig. 2. A uniform goal-plan tree of depth 2

is answered by deriving formulae that allow one to compute the number of behaviours, both successful and unsuccessful (i.e. failed), for a given goal-plan tree.

To allow us to perform the analysis we make the following *uniformity* assumptions about the form of the goal-plan tree, as illustrated in Figure 2.

1. We assume that all subtrees of a goal or plan node have the same structure. We can therefore define the *depth* of a goal-plan tree as the number of layers of goal nodes it contains. A goal-plan tree of depth 0 is a plan with no sub-goals, while a goal-plan tree of depth $d > 0$ is either a plan node with children that are goal nodes at depth d, or a goal node with children that are plan nodes at depth $d-1$.
2. We assume that all plan instances at depth $d > 0$ have k sub-goals.
3. We assume that all goals have j applicable plan instances. This can be the case if each goal has j relevant plans, each of which results in exactly one applicable plan instance, but can also be the case in other ways, for instance a goal may have $2j$ relevant plans, half of which are applicable in the current situation. Note that since we deal with applicable plans, we don't model context conditions.

Our analysis uses the following terminology:

- Our uniformity assumptions mean that the structure of the subtree rooted at a goal or plan node is determined solely by its depth, and we can therefore denote a goal or plan node at depth d as g_d or p_d (respectively).
- We use $n^{\checkmark}(x_d)$ to denote the number of *successful* execution paths of a goal-plan tree of depth d rooted at x (where x is either a goal g or a plan p). Where specifying d is not important we will sometimes elide it, writing $n^{\checkmark}(x)$.
- Similarly, we use $n^{\times}(x_d)$ to denote the number of *unsuccessful* execution paths of a goal-plan tree of depth d with root x (either g or p).
- We extend this notation to sequences $x_1;\ldots;x_n$ where each x_i is a goal or action and ';' denotes sequential composition. We abbreviate a sequence of n occurrences of x by x^n.

4.1 Base Case: Successful Executions

We begin by calculating the number of *successful* paths through a goal-plan tree in the absence of failure (and of failure handling). This analysis follows that of Padgham and Winikoff [10, pages 17–19].

Roughly speaking, for a goal, the number of ways it can be achieved is the *sum* of the number of ways in which its children can be achieved (since the children represent alternatives, i.e. "or"). On the other hand, for a plan, the number of ways it can be achieved is the *product* of the number of ways in which its children can be achieved (since the children must all be achieved, i.e. "and").

Given a tree with root g (a goal), assume that each of its j children can be achieved in n different ways[5], then, because we select one of the children[6], the number of ways in which g can be achieved is jn. Similarly, for a tree with root p (a plan), assume that each of its k children can be achieved in n different ways, then, because we execute all of its children, the number of ways in which p can be executed is $n \cdots n$, or n^k. A plan with no children (i.e. at depth $d = 0$) can be executed (successfully) in exactly one way. This yields the following definition:

$$\begin{aligned} n^{\checkmark}(g_d) &= j\,n^{\checkmark}(p_{d-1}) \\ n^{\checkmark}(p_0) &= 1 \\ n^{\checkmark}(p_d) &= n^{\checkmark}({g_d}^k) = n^{\checkmark}(g_d)^k \end{aligned}$$

If $k = 1$ we have $n^{\checkmark}(g_d) = n^{\checkmark}(p_d) = j^d$. If $k > 1$, we obtain the following definitions:

$$n^{\checkmark}(g_d) = j^{\sum_{i=0}^{d-1} k^i} = j^{(k^d-1)/(k-1)} \tag{1}$$

$$n^{\checkmark}(p_d) = (j^{\sum_{i=0}^{d-1} k^i})^k = j^{k(k^d-1)/(k-1)} \tag{2}$$

4.2 Adding Failure

We now extend the analysis to include failure, and determine the number of *unsuccessful* executions, i.e. executions that result in failure of the top-level goal. For the moment we assume that there is no failure handing (we add failure handling in Section 4.3).

In order to determine the number of failed executions we have to know where failure can occur. In BDI systems there are two places where failure occurs: when a goal has no applicable plan instances, and when an action or goal within a plan body fails. However, our uniformity assumptions mean that we do not address the former case—it is assumed that a goal will always have j instances of applicable plans. Note that this is a *conservative* assumption: relaxing it results in the number of unsuccessful executions being even larger.

In the case of an action failing, this causes the plan that contains it to fail, which will (in the absence of failure handling) cause its parent goal node to fail. A plan can therefore fail due to the failure of any of the actions and subgoals that are sequentially composed within its body. More specifically, a plan fails if an attempt to sequentially execute its body fails, so we begin by considering how the sequential composition of an action or goal with a sequence of further actions or goals can succeed and fail. The sequence $x;s$ can succeed if both x and s succeed, i.e. $n^{\checkmark}(x;s) = n^{\checkmark}(x)\,n^{\checkmark}(s)$. The sequence fails if

[5] Because the tree is assumed to be uniform, all of the children can be achieved in the same number of ways, allowing us to write jn rather than $n_1 + \ldots + n_j$.

[6] We make no assumption that plans are selected in a given order.

either x fails, or if x succeeds but s fails. Therefore we have $n^{\text{✘}}(x;s) = n^{\text{✘}}(x) + n^{\text{✔}}(x)\,n^{\text{✘}}(s)$. It follows that $n^{\text{✔}}(x^k) = n^{\text{✔}}(x)^k$ and $n^{\text{✘}}(x^k) = n^{\text{✘}}(x)\,(1 + \cdots + n^{\text{✔}}(x)^{k-1})$, which can easily be proved by induction.

More generally, we assume there are ℓ actions before, after, and between the subgoals in a plan, i.e. the body of a plan has the form $a^\ell;(g_d;a^\ell)^k$. A plan with no subgoals (i.e. at depth 0) is considered to consist of ℓ actions (which is quite conservative: in particular, $\ell = 1$ implies that plans at depth 0 consist of a single action).

We can then give the following definitions for the number of *unsuccessful* executions of a goal-plan tree, *without* failure handling. The equation for $n^{\text{✘}}(g_d)$ is derived using the same reasoning as in the previous section: a single plan is selected and executed, and there are j plans. The second line of the definition of $n^{\text{✘}}(p_d)$ is obtained by expanding the first line using the definitions of $n^{\text{✔}}$ and $n^{\text{✘}}$ for sequential execution.

$$n^{\text{✘}}(g_d) = j\,n^{\text{✘}}(p_{d-1}) \tag{3}$$

$$n^{\text{✘}}(p_0) = \ell \tag{4}$$

$$\begin{aligned} n^{\text{✘}}(p_d) &= n^{\text{✘}}(a^\ell;(g_d;a^\ell)^k) \\ &= \ell + \left(n^{\text{✘}}(g_d) + \ell\, n^{\text{✔}}(g_d)\right) \frac{n^{\text{✔}}(g_d)^k - 1}{n^{\text{✔}}(g_d) - 1} \quad (\text{for } d > 0 \text{ and } n^{\text{✔}}(g_d) > 1) \end{aligned} \tag{5}$$

Finally, we note that the analysis of the number of *successful* executions of a goal-plan tree in the absence of failure handling presented in Section 4.1 is unaffected by the addition of actions to plan bodies. This is because there is only one way for a sequence of actions to succeed, so Equations 1 and 2 remain correct.

4.3 Adding Failure Handling

We now consider how the introduction of a failure-handling mechanism affects the analysis. A common means of dealing with failure in BDI systems is to respond to the failure of a plan by trying an alternative applicable plan for the event that triggered that plan. For example, suppose a goal g has three applicable plans *pa*, *pb* and *pc*; *pa* is selected, and it fails. Then the failure-handling mechanism will respond by selecting *pb* or *pc* and executing it. Assume that *pc* is selected. Then if *pc* fails, the last remaining plan (*pb*) is used, and if it too fails, then the goal is deemed to have failed.

The result of this is that, as we might hope, it is *harder* to fail: the only way a goal execution can fail is if *all* of the applicable plans are tried and *each* of them fails[7].

The number of executions can then be computed as follows: if the goal has j applicable plan instances $p1,\ldots,pj$ and each of the plans has $n_i = n^{\text{✘}}(pi)$ unsuccessful executions, then we have $n_1 \cdots n_j$ unsuccessful executions of all of the plans. Since the plans can be selected in any order we multiply this by $j!$ yielding the following definition.

$$n^{\text{✘}}(g_d) = j!\,n^{\text{✘}}(p_{d-1})^j \tag{6}$$

[7] As noted earlier, our uniformity assumptions imply that goals cannot fail due to a lack of applicable plan instances. Allowing for this additional type of failure would result in the number of behaviours being even larger.

Params.			Number of			Without failure handling		With failure handling	
j	k	d	goals	plans	actions	$n^{\checkmark}(g)$	$n^{\times}(g)$	$n^{\checkmark}(g)$	$n^{\times}(g)$
2	2	3	21	42	62 (13)	128	614	$\approx 6.33\times10^{12}$	$\approx 1.82\times10^{13}$
3	3	3	91	273	363 (25)	1,594,323	6,337,425	$\approx 1.02\times10^{107}$	$\approx 2.56\times10^{107}$
2	3	4	259	518	776 (79)	1,099,511,627,776	6,523,509,472,174	$\approx 1.82\times10^{157}$	$\approx 7.23\times10^{157}$
3	4	3	157	471	627 (41)	10,460,353,203	41,754,963,603	$\approx 3.13\times10^{184}$	$\approx 7.82\times10^{184}$

Table 1. Illustrative values for $n^{\checkmark}(g)$ and $n^{\times}(g)$ with and without failure handling

Because failure handling happens at the level of goals, the number of ways in which a *plan* can fail is still defined by Equations 4 and 5, but with $n^{\times}(g)$ referring to the new definition in Equation 6.

Turning now to the number of *successful* executions (i.e. $n^{\checkmark}(x)$) we observe that the effect of adding failure handling is to *convert failures to successes*, i.e. an execution that would otherwise be unsuccessful is extended into a longer execution that may succeed. A successful execution of a goal may involve a number of failed attempts to execute different applicable plans, followed by a successful execution of another applicable plan. We assume that the applicable plans could be attempted in any order[8].

We can then derive the following equations (again, since failure handling is done at the goal level, the analysis for plans is the same as in Section 4.1):

$$n^{\checkmark}(g_d) = \sum_{i=0}^{j-1} \binom{j}{i} i!\, n^{\times}(p_{d-1})^{i}\, (j-i)\, n^{\checkmark}(p_{d-1}) \tag{7}$$

$$n^{\checkmark}(p_0) = 1 \tag{8}$$

$$n^{\checkmark}(p_d) = n^{\checkmark}(g_d)^{k} \ \text{(for } d > 0\text{)} \tag{9}$$

The equation for $n^{\checkmark}(g_d)$ can be explained as follows. Each successful execution of a goal involves a sequence of i plan executions that fail (for some i, $0 \le i < j$) followed by one plan execution that succeeds. There are $\binom{j}{i}$ ways of choosing the failed plans, which can be ordered in $i!$ ways, and each plan has $n^{\times}(p_{d-1})$ ways to fail. There are then $j-i$ ways of choosing the final successful plan, which has $n^{\checkmark}(p_{d-1})$ ways to succeed.

Table 1 makes the various equations developed so far concrete by showing illustrative values for $n^{\times}$ and $n^{\checkmark}$ for a range of reasonable (and fairly low) values for j, k and d and using $\ell = 1$. The "Number of" columns show the number of goals and plans in the tree, and the number of actions in the tree. The number of actions in brackets is how many actions are executed in a single (successful) execution with no failure handling.

4.4 The Probability of Failing

In Section 4.3 we said that introducing failure handling makes it harder to fail. However, Table 1 appears at first glance to contradict this, in that there are many more ways of failing with failure handling than there are without failure handling.

[8] When an applicable plan for a goal fails, we assume that an updated set of applicable plans is computed (excluding those already tried for the current goal). We assume the cardinality of this set decreases by one each time, and we model non-determinism due to side effects of failed actions by considering that the remaining plans could be generated in any order.

The key to understanding the apparent discrepancy is to consider the *probability* of failing: Table 1 merely counts the number of possible execution paths, without considering the likelihood of a particular path being taken. Working out the probability of failing (as we do below) shows that although there are many more ways of failing (and also of succeeding), the probability of failing is, indeed, much lower.

Let us denote the probability of an execution of a goal-plan tree with root x and depth d failing as $p^{\text{✘}}(x_d)$, and the probability of it succeeding as $p^{\text{✔}}(x_d) = 1 - p^{\text{✘}}(x_d)$.

We assume that the probability of an action failing is ε_a[9]. Then the probability of a given plan's actions all succeeding is simply $(1-\varepsilon_a)^x$ where x is the number of actions. Hence the probability of a plan failing due to the failure of (one of) its actions is simply $1-(1-\varepsilon_a)^x$, i.e. for a plan at depth 0 the probability of failure is $\varepsilon_0 = 1-(1-\varepsilon_a)^\ell$ and for a plan at depth greater than 0 the probability of failure due to actions is $\varepsilon = 1-(1-\varepsilon_a)^{\ell(k+1)}$ (recall that such a plan has ℓ actions before, after, and between, each of its k sub-goals). Considering not only the actions but also the sub-goals $g_1,\ldots,g_k$ of a plan p, we have that for the plan to succeed, all of the sub-goals must succeed, and additionally, the plan's actions must succeed giving $p^{\text{✔}}(p_d) = (1-\varepsilon)\,p^{\text{✔}}(g_d)^k$. We can easily derive from this an equation for $p^{\text{✘}}(p_d)$ (given below). Note that the same reasoning applies to a plan regardless of whether there is failure handling, because failure handling is done at the goal level.

In the absence of failure handling, for a goal g with possible plans $p_1,\ldots,p_j$ to succeed we select one plan and execute it, so the probability of success is the probability of that plan succeeding, i.e. $p^{\text{✔}}(g_d) = p^{\text{✔}}(p_{d-1})$. Here we ignore the possibility of a goal failing due to there being no applicable plans. Extending the analysis to include this possibility does not change the results significantly [15].

Formally, then, we have for the case without failure handling:

$$\begin{aligned} p^{\text{✘}}(g_d) &= p^{\text{✘}}(p_{d-1}) \\ p^{\text{✘}}(p_0) &= \varepsilon_0 \\ p^{\text{✘}}(p_d) &= 1-[(1-\varepsilon)\,(1-p^{\text{✘}}(g_d))^k] \end{aligned}$$

Now consider what happens when failure handling is added. In this case, in order for a goal to fail, *all* of the plans must fail. Since failure is handled at the goal level, the only change to the above equations is the following:

$$p^{\text{✘}}(g_d) = p^{\text{✘}}(p_{d-1})^j$$

It is not easy to see from the equations what the patterns of probabilities actually are, and so, for illustration purposes, Table 2 shows what the probability of failure is, both with and without failure handling, for two scenarios. These values are computed using $j = k = 3$ (i.e. a relatively small branching factor) and with $\ell = 1$. We consider two cases: where $\varepsilon_a = 0.05$ and hence $\varepsilon \approx 0.185$ (which is rather high); and where $\varepsilon_a = 0.01$ and hence $\varepsilon \approx 0.04$.

As can be seen, without failure handling, failure is *magnified*: the larger the goal-plan tree is, the more actions are involved, and hence the greater the chance of an action

[9] For simplicity, we assume that the failure of an action in a plan is independent of the failure of other actions in the plan.

ε_a	d	No failure handling	With failure handling
0.05	2	30%	0.64%
	3	72%	0.81%
	4	98%	0.86%
0.01	2	7%	0.006%
	3	22%	0.006%
	4	55%	0.006%

Table 2. Probabilities of failure

somewhere failing, leading to the failure of the top-level goal (since there is no failure handling). On the other hand, with failure handling, the probability of failure is low and doesn't appear to grow significantly as the goal-plan tree grows.

4.5 Bounding the Number of Action Failures

In this section we briefly examine how the number of traces for a goal-plan tree is affected if a bound is put on the number of action failures that can occur. We define $n^{\checkmark}_{caf}(g_d, n)$ and $n^{\times}_{caf}(g_d, n)$ to be the numbers of successful and failed (respectively) executions of a goal at depth d in which *at most n* action failures have occurred (the *caf* subscript abbreviates "cumulative action failures"). We are interested in computing the sequences $\{n^{\checkmark}_{caf}(g_d, n)\}_{n=0}^{\infty}$ and $\{n^{\times}_{caf}(g_d, n)\}_{n=0}^{\infty}$. We have derived a set of equations that define a recursive procedure for computing the *ordinary generating functions* [16] for these sequences for a given value of d[10]. The Sage mathematical software system[11] was used to obtain the power series representations of these functions (which are polynomial functions of finite order) for particular values of d, j, k and l. The coefficients of these polynomials then gave us the values of $n^{\checkmark}_{caf}(g_d, n)$ and $n^{\times}_{caf}(g_d, n)$ for each n. We found that even with low bounds on the number of action failures, large numbers of traces were still generated. For example, for $j = k = 2$, $\ell = 1$, $d = 3$ and five action failures allowed, there were still 1,186,693,266 possible traces in total. For $j = k = d = 3$, $\ell = 1$ and ten action failures allowed, the total number of traces exceeded 4×10^{36}. Details can be found in the full version of this paper [15].

5 A Reality Check

In the previous section we analysed an abstract model of BDI execution in order to determine the size of the behaviour space. The analysis yielded information about the size of the behaviour space and how it is affected by various factors, and on the probability of a goal failing.

In this section we consider the issue of whether this analysis is applicable to real systems. The analysis made a number of simplifying assumptions, and these mean that the results may not apply to real systems. In order to assess to what extent our analysis applies to real systems we compared our abstract BDI execution model with results from

[10] The equations and their derivations are beyond the scope of this paper.

[11] http://www.sagemath.org/

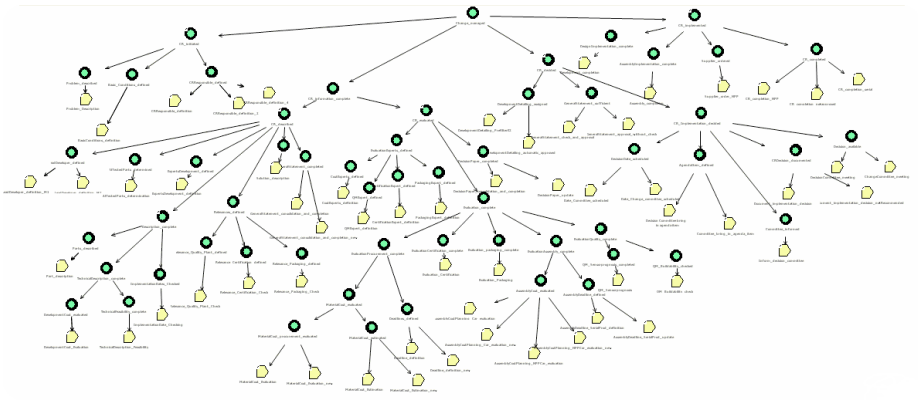

Fig. 3. Goal-plan tree ([17, Figure 6], copied with permission from IFAAMAS)

a real BDI platform (namely JACK[12] [12]). This comparison allows us to assess to what extent the analysis of our abstract BDI execution model matches the execution that takes place in a real (industrial strength) BDI platform. We mapped example goal-plan trees to JACK, using a test harness that allowed us to explore all possible executions. The results matched those computed by the Prolog code of Figure 1, giving precisely the same traces for each of the trees. This indicates that our abstract BDI execution model is indeed an accurate description of what takes place in a real BDI platform (specifically JACK).

We also analysed a goal-plan tree from a real industrial application. This analysis allows us to determine the extent to which the conclusions of our analysis of uniform goal-plan trees applies to real applications, where the goal-plan trees are not likely to be uniform. In other words, to what extent do the large numbers in Table 1 apply to real applications?

As an example of a real application we consider a recent industrial application at Daimler which used BDI agents to realise agile business processes [17]. Figure 3 shows[13] a goal-plan tree from Burmeister et al. [17] which has "*60 achieve goals in up to 7 levels. 10 maintain goals, 85 plans and about 100 context variables*" [17, Page 41]. Unlike the typical goal-plan trees used in BDI platforms, the tree in Figure 3 consists of layers of "and"-refined goals, with the only "or" refinements being at the leaves (where the plans are). In terms of the analysis presented in this paper we can treat a link from a goal g to a set of goals, say, g_1, g_2, g_3 as being equivalent to the goal g having a single plan p which performs g_1, g_2, g_3 (and has no actions, i.e. $\ell = 0$ for non-leaf plans).

The last row of Table 3 gives the various n values for this goal-plan tree, for $\ell = 4$ (top row), $\ell = 2$ (middle row) and $\ell = 1$ (bottom row). Note that these figures are actually *lower bounds* because we assumed that plans at depth 0 are simple linear combina-

[12] JACK is a descendent of a line of BDI platforms going back to PRS, which makes it a good representative for a whole family of BDI platforms.

[13] The details are not meant to be legible: the structure is what matters.

Parameters			Number of		Without failure handling (Secs. 4.1 and 4.2)		With failure handling (Section 4.3)	
j	k	d	goals	actions	$n^{\checkmark}(g)$	$n^{\times}(g)$	$n^{\checkmark}(g)$	$n^{\times}(g)$
2	2	3	21	62 (13)	128	614	$\approx 6.33 \times 10^{12}$	$\approx 1.82 \times 10^{13}$
3	3	3	91	363 (25)	1,594,323	6,337,425	$\approx 1.02 \times 10^{107}$	$\approx 2.56 \times 10^{107}$
Workflow with 57 goals*					294,912	3,250,604 ($\ell = 4$)	$\approx 2.98 \times 10^{20}$	$\approx 9.69 \times 10^{20}$
(* The paper says 60 goals,					294,912	1,625,302 ($\ell = 2$)	$\approx 6.28 \times 10^{15}$	$\approx 8.96 \times 10^{15}$
but Figure 3 has 57 goals)					294,912	812,651 ($\ell = 1$)	$\approx 9.66 \times 10^{11}$	$\approx 6.27 \times 10^{11}$

Table 3. Illustrative values for $n^{\checkmark}(g)$ and $n^{\times}(g)$

tions of ℓ actions, whereas it is clear from Burmeister et al. [17] that their plans are in fact more complicated, and can contain nested decision making [17, Figure 4].

A rough indication of the size of a goal-plan tree is the number of goals. With 57 goals, the tree of Figure 3 has size in between the first two rows of Table 3. Comparing the number of possible behaviours of the uniform goal-plan trees against the non-uniform, but real, goal-plan tree, we see that the behaviour space is somewhat smaller in the non-uniform tree, but that it is still quite large, especially in the case with failure handling. However, we do need to remember (a) that the tree of Figure 3 only has plans at the leaves, which reduces its complexity; and (b) that the figures for the tree are a conservative estimate, since we assume that leaf plans have only simple behaviour.

6 Conclusion

To summarise, our analysis has found that the space of possible behaviours for BDI agents is, indeed, large. As expected, the number of possible behaviours grows as the tree's depth (d) and breadth (j and k) grow. However, somewhat surprisingly, the introduction of failure handling makes a very significant difference to the number of behaviours. For instance, adding failure handling for a uniform goal-plan tree with $d = 3$ and $j = k = 2$ took the number of successful behaviours from 128 to 6,332,669,231,104.

Before we consider the negative consequences of our analysis, it is worth highlighting one positive consequence: our analysis provides quantitative support for the long-held belief that BDI agents allow for the definition of highly flexible and robust agents. The number of behaviours supports the claim of flexibility, and the analysis of the probability of failure supports the claim of robustness.

So what does the analysis in this paper tell us about the testability of BDI agent systems? The behaviour space sizes depicted in Tables 1 and 3 suggest quite strongly that attempting to obtain assurance of a system's correctness by testing the system as a whole is not feasible. Considering also the probability of failure, we observe that the space of *unsuccessful* executions is particularly hard to test, since there are many unsuccessful executions (more than successful ones), and the probability of an unsuccessful execution is low, making this part of the behaviour space hard to "reach". With respect to unit and integration testing we observe that it is not always clear how to usefully test parts of a complex and possibly emergent system. For example, given an ant colony optimisation system, testing a single ant doesn't provide much useful information about the correct functioning of the whole system.

We acknowledge that our analysis is somewhat pessimistic: real BDI systems do not necessarily have deep or heavily branching goal-plan trees. Indeed, the tree described in Section 5 has a smaller behaviour space than the abstract goal-plan trees analysed in Section 4. However, it is still quite large, and this did cause problems in validation [17].

One possible recommendation is to avoid using BDI features that lead to a large behaviour space, but this would mean keeping BDI goal-plan trees shallow and sparse and/or avoiding failure handling, both of which are key benefits of the approach.

Another approach is to accept that, at present, there is no practical way of assuring that BDI agents will be behave appropriately in all possible situations. It is worth noting that humans are similar in this respect: there is no way of assuring that (e.g.) a pilot or surgeon will behave appropriately in all situations, and so safety precautions are built in to certain systems (e.g. a process that double-checks information, or a backup system such as a co-pilot).

There are two areas of work that may, in the future, yield viable approaches for the assurance of BDI agents. One is to explore techniques for exploiting regularities in the behaviour space (analogous to state space slicing techniques used in model checking). Another is continuing to develop formal methods for agents, which are promising, but still not really usable for large systems. Space precludes a detailed discussion of formal methods, and so we refer the reader to a recent book for an overview of current work [18], including a chapter on the role of formal methods in the assurance of agents [19].

Other future work includes extending the analysis in various ways, e.g. to cover multiple agents using a notion of distributed goal-plan trees; deal with types of goals other than achievement goals; and address other types of agents. Additionally, the analysis could be extended to model the environment and resources. Our analysis considered a real application, but clearly it would be desirable to consider a *range* of applications. This could provide additional evidence that the analysis is not unduly pessimistic, and would also lead to an understanding of the *variance* in goal-plan trees and their characteristics across applications. Finally, as noted earlier, there is clearly work to be done in improving testing techniques, and in continuing to develop formal methods for MAS (e.g. [20, 21]).

References

1. Munroe, S., Miller, T., Belecheanu, R., Pěchouček, M., McBurney, P., Luck, M.: Crossing the agent technology chasm: Experiences and challenges in commercial applications of agents. Knowledge Engineering Review **21**(4), 345–392 (2006)
2. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: J. Allen, R. Fikes, E. Sandewall (eds.) Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference, pp. 473–484. Morgan Kaufmann (1991)
3. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
4. Benfield, S.S., Hendrickson, J., Galanti, D.: Making a strong business case for multiagent technology. In: P. Stone, G. Weiss (eds.) Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 10–15. ACM Press (2006)

5. Naish, L.: Resource-oriented deadlock analysis. In: V. Dahl, I. Niemelä (eds.) Logic Programming, *Lecture Notes in Computer Science*, vol. 4670, pp. 302–316. Springer, Berlin/Heidelberg (2007)
6. Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: J. Filipe, B. Shishkov, M. Helfert, L. Maciaszek (eds.) Software and Data Technologies, *Communications in Computer and Information Science*, vol. 22, pp. 399–413. Springer, Berlin/Heidelberg (2009)
7. Ekinci, E.E., Tiryaki, A.M., Çetin, Ö., Dikenelli: Goal-oriented agent testing revisited. In: M. Luck, J.J. Gomez-Sanz (eds.) Agent-Oriented Software Engineering IX, *Lecture Notes in Computer Science*, vol. 5386, pp. 173–186. Springer, Berlin/Heidelberg (2009)
8. Gomez-Sanz, J.J., Botía, J., Serrano, E., Pavón, J.: Testing and debugging of MAS interactions with INGENIAS. In: M. Luck, J.J. Gomez-Sanz (eds.) Agent-Oriented Software Engineering IX, *Lecture Notes in Computer Science*, vol. 5386, pp. 199–212. Springer, Berlin/Heidelberg (2009)
9. Nguyen, C.D., Perini, A., Tonella, P.: Experimental evaluation of ontology-based test generation for multi-agent systems. In: M. Luck, J.J. Gomez-Sanz (eds.) Agent-Oriented Software Engineering IX, *Lecture Notes in Computer Science*, vol. 5386, pp. 187–198. Springer, Berlin/Heidelberg (2009)
10. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons (2004)
11. Shaw, P., Farwer, B., Bordini, R.: Theoretical and experimental results on the goal-plan tree problem. In: Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1379–1382. IFAAMAS (2008)
12. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News (2) (1999). http://www.agentlink.org/newsletter/2/newsletter2.pdf
13. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: W.V. de Velde, J. Perrame (eds.) Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), *Lecture Notes in Artificial Intelligence*, vol. 1038, pp. 42–55. Springer, Berlin/Heidelberg (1996)
14. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), pp. 470–481. Morgan Kaufmann, Toulouse, France (2002)
15. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. Discussion Paper 2008/03, Department of Information Science, University of Otago (2008). http://eprints.otago.ac.nz/793/
16. Wilf, H.S.: generatingfunctionology, second edn. Academic Press Inc., Boston, MA (1994). http://www.math.upenn.edu/~wilf/gfology2.pdf
17. Burmeister, B., Arnold, M., Copaciu, F., Rimassa, G.: BDI-agents for agile goal-oriented business processes. In: Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 37–44. IFAAMAS (2008)
18. Dastani, M., Hindriks, K.V., Meyer, J.J.C. (eds.): Specification and Verification of Multi-agent systems. Springer, Berlin/Heidelberg (2010)
19. Winikoff, M.: Assurance of Agent Systems: What Role should Formal Verification play?, chap. 12, pp. 353–383. In: Dastani et al. [18] (2010)
20. Raimondi, F., Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. Journal of Applied Logic **5**(2), 235–251 (2007)
21. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 409–416. ACM Press (2003)