On the testability of BDI agent systems

Michael Winikoff · Stephen Cranefield

Michael Winikoff
University of Otago
Tel.: +64 3 479 8386
Fax: +64 3 479 8311
E-mail: Michael.Winikoff@otago.ac.nz

Stephen Cranefield
University of Otago
Tel.: +64 3 479 8083
Fax: +64 3 479 8311
E-mail: Stephen.Cranefield@otago.ac.nz

Abstract Before deploying a software system we need to assure ourselves (and stakeholders) 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 "obvious 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, the factors that influence the size are not what we expected them to be; specifically, 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.

Keywords Testing · Complexity · Validation · Belief-Desire-Intention (BDI)

1 Introduction

Increasingly we are called upon to develop software systems that operate in dynamic environments, that are robust in the face of failure, that are required to exhibit flexible behaviour, and that operate in open environments. One approach for developing such systems that has demonstrated its effectiveness in a range of domains is the use of the metaphor of *software agents* to conceptualise, design and build software systems [1]. Agent systems have been increasingly finding deployment in a wide range of applications (e.g. [2,3]).

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:

> "*. . . validation 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 . . .*" [2, 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) [4, 5] approach to realising adaptive and flexible agents, which has been demonstrated to be practically applicable, resulting in reduced development cost and increased flexibility [3].

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 well-known).

Specifically, we aim to answer these questions:

1. How large is the behaviour space for BDI agents?
2. What factors influence the size of the behaviour space?
3. Is it feasible to assure the effectiveness of BDI systems by testing?

Let us briefly consider the relationship between the feasibility of testing and the behaviour space size. Running a test consists of examining a single execution trace and determining whether it is correct or not. Determining whether an execution is correct can be done in a range of ways (e.g. consulting an oracle or looking for certain conditions whose violation indicates an error). The concern of this paper is not how to check whether a given trace is correct, but rather the feasibility of assuring correctness by testing. This feasibility boils down to the number of traces: roughly speaking, our confidence in a system corresponds to the proportion of its traces that have been tested (and found to be correct). Thus, if there are many traces, it will require more tests to obtain a given level of confidence.

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 (see, for example [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 and Bordini [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.

Other related work concerns Hierarchical Task Network (HTN) planning [12]. BDI agents are equipped with pre-supplied plans that must be generated either by human domain experts and/or by an automated planning process. While there has been little work

in automated planning for BDI agents, there is a clear similarity between HTN plans and BDI programs [13], in that both use a hierarchical representation with goals ("non-primitive tasks" in HTN terminology), plans ("decomposition methods") and goal-plan trees ("task networks"). The HTN plan existence problem (does a plan exist?) has been studied, and in settings that correspond to BDI execution (many goals, total ordering within plans, and with variables) is known to be EXPSPACE-hard and in DEXPTIME [12, 14]. However, this work does not appear to provide any answers to the above questions. This is due to a significant difference between HTN planning and BDI program execution. Although selecting a BDI plan to solve a goal corresponds to decomposing an HTN task using a method, the treatment of actions and the notion of failure differ between these two processes. A BDI agent will execute an atomic action that is the next step of the agent's active plan, possibly resulting in failure and the initiation of the BDI platform's failure-handling process. In contrast, a standard HTN planner does not execute any "primitive tasks"[1]. An HTN planner fails and backtracks to consider alternative task decomposition methods when the constraints introduced by the expansion of a task are incompatible with the existing network's constraints. These constraints include assertions that a given literal must be true immediately before or after a task is performed. Once backtracking occurs, the task network contains no record of any primitive tasks that may have been temporarily but unsuccessfully introduced by a method. This is valid as there is no notion of an action failing. Like most other planning techniques, HTN planning assumes that actions will always succeed and produce their specified effects when their preconditions[2] are true.

In contrast, we assume that every action in a BDI agent programme could potentially fail for a variety of reasons such as unexpected environment changes, resource limitations and incorrectly specified context conditions on plans. Furthermore, actions (failed or successful) cannot, in general, be undone. Therefore the traces that must be considered for testing coverage must include failed actions as well as the subsequent attempts to recover from the failure. This difference means that the complexity analysis of Erol *et al.* [12, 14] is of a different problem, and that our analysis of the number of failures in BDI execution (Section 4.2) is specific to BDI execution.

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

Before we describe the Belief-Desire-Intention (BDI) model we explain why we chose this model. In addition to being well known and widely-used, the BDI model is well defined and generic. That it is well defined allows us to analyse the behaviour spaces that result from using it. That it is generic implies that our analysis applies to a wide range of platforms.

[1] There are approaches that blur this difference by adding look-ahead planning to BDI or online execution to HTNs, for example the planner in the RETSINA multi-agent system [15] has the ability to interleave planning and execution. However, no theoretical analysis of this extension has been reported, and the analysis of Erol *et al.* [12, 14] applies to "classical" HTN planning.

[2] In the case of HTN planning, primitive task preconditions are encoded as constraints on task networks.

The BDI model can be viewed from philosophical [5] and logical [4] perspectives, but we are interested here in the *implementation* perspective, as exhibited in a range of architectures and platforms (such as JACK [16], JAM [17], dMARS [18], PRS [19,20], UM-PRS [21], Jason [22], SPARK [23], Jadex [24], IRMA [25] etc.). For the purposes of our analysis here, a formal and detailed presentation is unnecessary. Those interested in formal semantics for BDI languages are referred to (e.g.) [26,27,22].

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[3]. 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[4], however for our purposes (following abstract notations such as AgentSpeak(L) [26] and CAN [27]) we assume that a plan body is a sequence of steps, where each step is either an action[5] (which can succeed or fail) or an event 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.

For example, consider the simple plans shown in Figure 1. For the first plan, Plan A, the plan is relevant for handling the event "achieve goal go-home", and it is applicable in situations where the agent believes that a train is imminent. The plan body consists of a sequence of four steps (in this case we assume that these are actions, but they could also be modelled as events that are handled by further plans).

A key feature of the BDI approach is that each plan encapsulates the conditions under which it is applicable (by defining an event pattern and context condition) [28]. This allows for additional plans for a given event to be added in a modular fashion, since the invoking context (i.e. where the triggering event is posted) does not contain code that selects amongst the available plans, and this is a key reason for the flexibility of BDI programming.

A typical BDI execution cycle is an elaboration of the following event-driven process (summarised in Figure 2)[6]:

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. A relevant plan instance is applicable if its context

[3] Other types of event typically include the addition and removal of beliefs from the agent's belief set.

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

[5] 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.

[6] 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.

Plan A: **handles event**: *achieve goal* go-home
context condition: train imminent
plan body:
(1) walk to train station
(2) check train running on time
(3) catch train
(4) walk home

Plan B: **handles event**: *achieve goal* go-home
context condition: not raining and have bicycle
plan body:
(1) cycle home

Plan C: **handles event**: *achieve goal* go-home
context condition: true (i.e. always applicable)
plan body:
(1) walk to bus stop
(2) check buses running
(3) catch bus
(4) walk home

Fig. 1 Three Simple Plans

```
Boolean function execute(event)
let relevant-plans = set of plan instances resulting from
      matching all plans' event patterns to event
let tried-plans = ∅
while true do
      let applicable-plans = set of plan instances resulting from
            solving the context conditions of relevant-plans
      applicable-plans := applicable-plans \ tried-plans
      if applicable-plans is empty then return false
      select plan p ∈ applicable-plans
      tried-plans := tried-plans ∪ {p}
      if execute(p.body) = true then return true
endwhile

Boolean function execute(plan-body)
if plan-body is empty then return true
elseif execute(first(plan-body)) = false then return false
else return execute(rest(plan-body))
endif

Boolean function execute(action)
attempt to perform the action
if action executed successfully then return true else return false endif
```

Fig. 2 BDI Execution Cycle

condition is true. 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 assump-

tions 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.

Regarding the final step, 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 re-calculate the set of applicable plans (ignoring those that have already been tried), as is done in Figure 2. 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.

One alternative failure-handling approach, used by Jason [22], is to post a failure event that can be handled by a user-provided plan. Although this is more flexible, since the user can specify what to do upon failure, it does place the burden of specifying failure handling on the user. Note that Jason does provide a "pattern" that allows the traditional BDI failure-handling mechanism to be specified [22, pages 171–172]. Another alternative failure-handling approach is used by 2APL [29] and its predecessor, 3APL: they permit the programmer to write "plan repair rules" which conditionally rewrite a (failed) plan into another plan. This approach, like Jason's, is quite flexible, but is not possible to analyse in a general way because the plan rules can be quite arbitrary. Another well-known BDI architecture is IRMA [25], which is described at a high-level and does not prescribe a specific failure-handling mechanism:

> "A full development of this architecture would have to give an account of the ways in which a resource-bounded agent would monitor her prior plans in the light of changes in belief. However this is developed, there will of course be times when an agent will have to give up a prior plan in light of a new belief that this plan is no longer executable. When this happens, a new process of deliberation may be triggered . . . "

Given the BDI execution cycle discussed above, the three example plans given earlier (Figure 1) can give rise to a range of behaviours, including the following:

- Suppose the event "achieve goal go-home" is posted and the agent believes that a train is imminent. It walks to the train station, finds out that the train is running on time, catches the train, and then walks home.
- Suppose that upon arrival at the train station the agent finds out that trains are delayed. Step (2) of Plan A fails, and the agent considers alternative plans. If it is raining at the present time, then Plan B is not applicable, and so Plan C is adopted (to catch the bus).
- Suppose that the agent has decided to catch the bus (because no train is believed to be imminent, and it is raining), and that attempting to execute Plan C fails (e.g. there is a bus strike). The agent will reconsider its plans and if the rain has stopped (and it has a bicycle) it may then use Plan B.

In the previous section we observed that testing was in essence the process of running a system, and checking whether an observed behaviour trace is "correct" (i.e. conforms to a specification, which we do not model). In the context of BDI agents, where a behaviour trace is classified as being successful or failed, it is important to be aware that the correctness of a

given execution trace is independent of whether the trace is of a successful or failed execution. A successful execution may in fact 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 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, as summarised in Figure 2, 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[7] 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 3. 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):

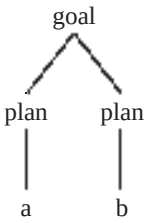
$$\begin{aligned}
\langle \mathit{GPT} \rangle &::= \texttt{goal([])} \mid \texttt{goal([}\langle \mathit{PlanList} \rangle\texttt{])} \\
\langle \mathit{PlanList} \rangle &::= \langle \mathit{Plan} \rangle \mid \langle \mathit{Plan} \rangle\texttt{,}\langle \mathit{PlanList} \rangle \\
\langle \mathit{Plan}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

For example, the simple goal-plan tree, corresponding to the following diagram, is modelled by the Prolog term: `goal([plan([act(a)]), plan([act(b)])]).`

[7] In order to be consistent with existing practice we shall use the term "goal" rather than "event" in the remainder of this paper.

```
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. 3 Prolog Code to Expand Goal-Plan Trees



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 3). When selecting a plan, we consider the possibility that any of the applicable plans could be chosen, not just the first plan. This is done because at different points in time different plan instances may be applicable. We saw an example of this earlier, where Plan A was chosen and failed, then Plan C was selected (and also failed), and then finally Plan B (which was not applicable when Plan A failed) was selected.

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 Figure 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. Again, this process can easily be seen to correspond to plan body execution in Figure 2.

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 do need to also 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). Again, this process can easily be seen to correspond to action execution in Figure 2. 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.

An example of applying this process to two example goal-plan trees can be found in the appendix.

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[8] 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, 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.

We make the following *uniformity* assumptions that allow us to perform the analysis. These simplifying assumptions concern the form of the goal-plan tree.

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.

Figure 4 shows a uniform goal-plan tree of depth 2.

We re-examine these assumptions in Section 4.4, where we allow the number of available plan instances (j) to vary across different levels of the tree, and in Section 5, where we consider a (non-uniform) goal-plan tree from an industrial application.

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 $n^{\checkmark}(x)$.

[8] We focus on a single goal in our analysis: multiple goals can be treated as the concurrent interleaving of the individual goals. Multiple agents can also be treated as concurrent interleaving, but some care needs to be taken with the details where an agent is waiting for another agent to respond.

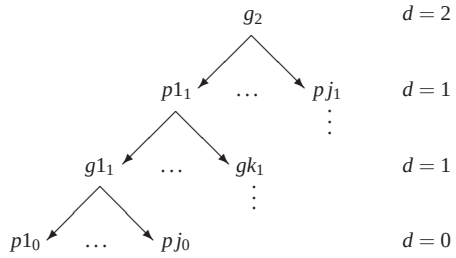


Fig. 4 A uniform goal-plan tree

- Similarly, we use $n^{\text{✗}}(x_d)$ 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 *plan body segments*, i.e. 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"). More precisely, $n^{\checkmark}(x_1;x_2) = n^{\checkmark}(x_1)\,n^{\checkmark}(x_2)$, that is, the sequence is successful if both x_1 and x_2 are successful.

Given a tree with root g (a goal), assume that each of its j children can be achieved in n different ways[9], then, because we select one of the children[10], 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:

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

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

[10] This is done in any order: in general there is no assumption that plans are selected in a given order.

Expanding this definition we obtain

$$
\begin{aligned}
n^{\checkmark}(g_1) &= j\, n^{\checkmark}(p_0) = j\,1 = j \\
n^{\checkmark}(g_2) &= j\, n^{\checkmark}(p_1) = j\,(n^{\checkmark}(g_1)^k) = j\,(j^k) = j^{k+1} \\
n^{\checkmark}(g_3) &= j\, n^{\checkmark}(p_2) = j\,(j^{k+1})^k = j^{k^2+k+1} \\
n^{\checkmark}(g_4) &= j\, n^{\checkmark}(p_3) = j\,(j^{k^2+k+1})^k = j^{k^3+k^2+k+1}
\end{aligned}
$$

which can be generalised to:

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

If $k > 1$ this can be simplified using the equivalence $k^{i-1} + \ldots + k^2 + k + 1 = (k^i - 1)/(k-1)$ to give the following closed form definition:

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

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

If $k = 1$ we have $n^{\checkmark}(g_d) = n^{\checkmark}(p_d) = j^d$.

Note that the equation for $n^{\checkmark}(p_d)$ assumes that sub-goals are achieved sequentially. If they are executed in parallel then the number of options is higher, since we need to consider all possible interleavings of the sub-goals' execution. For example, suppose that a plan p_d has two sub-goals, $g1_d$ and $g2_d$, where each of the sub-goals has $n^{\checkmark}(g_d)$ successful executions, and each execution has l steps (we assume for ease of analysis that both execution paths have the same length). The number of ways of interleaving two parallel executions, each of length l is (see, e.g., [30, Section 3]):

$$\binom{2l}{l} = \frac{(2\,l)!}{(l!)\,(l!)}$$

and hence the number of ways of executing p_d with parallel execution of subgoals is:

$$n^{\checkmark}(p_d) = n^{\checkmark}(g_d)^2 \binom{2l}{l} = n^{\checkmark}(g_d)^2 \frac{(2\,l)!}{(l!)\,(l!)}$$

In the remainder of this paper we assume that the sub-goals of a plan are achieved sequentially, since this is the common case, and since it yields a lower figure which, as we shall see, is still large enough to allow for conclusions to be drawn.

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 (within an applicable plan instance) fails. However, our uniformity assumption means 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 order to model the latter case we need to extend our model of plans to encompass actions. For example, suppose that a plan has a body of the form $a1;ga;a2;gb;a3$ where ai are actions, ga and gb are sub-goals, and ";" denotes sequential execution. Then the plan has the following five cases of *unsuccessful* (i.e. failed) executions:

1. $a1$ fails
2. $a1$ succeeds, but then ga fails
3. $a1$ and ga succeed, but $a2$ fails
4. $a1$, ga, and $a2$ succeed, but then gb fails
5. $a1$, ga, $a2$ and gb succeed, but $a3$ fails

Suppose that ga can be executed *successfully* in $n^{\checkmark}(ga)$ different ways, then the third case corresponds to $n^{\checkmark}(ga)$ different failed executions: for each successful execution of ga, extend it by adding a failed execution of $a2$ (actions can only be executed in one way, i.e. $n^{\checkmark}(a) = 1$ and $n^{\times}(a) = 1$). Similarly, if gb has $n^{\checkmark}(gb)$ successful executions then the fifth case corresponds to $n^{\checkmark}(ga)\,n^{\checkmark}(gb)$ different failed executions. If ga can be *unsuccessfully* executed in $n^{\times}(ga)$ different ways then the second case corresponds to $n^{\times}(ga)$ different executions. Similarly, the fourth case corresponds to $n^{\checkmark}(ga)\,n^{\times}(gb)$ different executions. Putting this together, we have that the total number of *unsuccessful* executions for a plan p with body $a1;ga;a2;gb;a3$ is the sum of the above five cases:

$$1 \;+\; n^{\times}(ga) \;+\; n^{\checkmark}(ga) \;+\; n^{\checkmark}(ga)\,n^{\times}(gb) \;+\; n^{\checkmark}(ga)\,n^{\checkmark}(gb)$$

More formally, $n^{\times}(x_1;x_2) = n^{\times}(x_1) + n^{\checkmark}(x_1)\,n^{\times}(x_2)$, that is, the sequence can fail if either x_1 fails, or if x_1 succeeds but x_2 fails. It follows that $n^{\checkmark}(x^k) = n^{\checkmark}(x)^k$ and $n^{\times}(x^k) = n^{\times}(x)\,(1 + \cdots + n^{\checkmark}(x)^{k-1})$, which can easily be proven by induction.

More generally, we assume there are ℓ actions before, after, and between the sub-goals in a plan, i.e. the above example plan corresponds to $\ell = 1$, and the following plan body corresponds to $\ell = 2$: $a1;a2;g3;a4;a5;g6;a7;a8$. A plan with no sub-goals (i.e. at depth 0) is considered to consist of ℓ actions (which is quite conservative: in particular, when we use $\ell = 1$ we assume that plans at depth 0 consist of only a single action).

The number of *unsuccessful* execution traces of a goal-plan tree can then be defined, based on the analysis above, as follows. First we calculate the numbers of successes and failures of the following repeated section of a plan body: $g_d;a^{\ell}$:

$$\begin{aligned}
n^{\checkmark}(g_d;a^{\ell}) &= n^{\checkmark}(g_d)\,n^{\checkmark}(a^{\ell}) \\
&= n^{\checkmark}(g_d)\,n^{\checkmark}(a)^{\ell} \\
&= n^{\checkmark}(g_d)\,1^{\ell} \\
&= n^{\checkmark}(g_d) \\
n^{\times}(g_d;a^{\ell}) &= n^{\times}(g_d) + n^{\checkmark}(g_d)\,n^{\times}(a^{\ell}) \\
&= n^{\times}(g_d) + n^{\checkmark}(g_d)\,n^{\times}(a)\,(1 + \cdots + n^{\checkmark}(a)^{\ell-1}) \\
&= n^{\times}(g_d) + n^{\checkmark}(g_d)\,\ell
\end{aligned}$$

We then have for $d > 0$:

$$\begin{aligned}
n^{\text{✘}}(p_d) &= n^{\text{✘}}(a^\ell; (g_d; a^\ell)^k) \\
&= n^{\text{✘}}(a^\ell) + n^{\text{✔}}(a^\ell)\, n^{\text{✘}}((g_d; a^\ell)^k) \\
&= n^{\text{✘}}(a)\,(1 + \cdots + n^{\text{✔}}(a)^{\ell-1})) + n^{\text{✔}}(a)^\ell\, n^{\text{✘}}((g_d; a^\ell)^k) \\
&= \ell + 1\, n^{\text{✘}}(g_d; a^\ell)\,(1 + \cdots + n^{\text{✔}}(g_d; a^\ell)^{k-1}) \\
&= \ell + (n^{\text{✘}}(g_d) + n^{\text{✔}}(g_d)\,\ell)\,(1 + \cdots + n^{\text{✔}}(g_d)^{k-1})) \\
&= \ell + \left(n^{\text{✘}}(g_d) + \ell\, n^{\text{✔}}(g_d)\right) \frac{n^{\text{✔}}(g_d)^k - 1}{n^{\text{✔}}(g_d) - 1} \quad \text{(assuming } n^{\text{✔}}(g_d) > 1\text{)}
\end{aligned}$$

This yields 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.

$$\begin{aligned}
n^{\text{✘}}(g_d) &= j\, n^{\text{✘}}(p_{d-1}) \\
n^{\text{✘}}(p_0) &= \ell \\
n^{\text{✘}}(p_d) &= \ell + \left(n^{\text{✘}}(g_d) + \ell\, n^{\text{✔}}(g_d)\right) \frac{n^{\text{✔}}(g_d)^k - 1}{n^{\text{✔}}(g_d) - 1} \\
&\qquad \text{(for } d > 0 \text{ and } n^{\text{✔}}(g_d) > 1\text{)}
\end{aligned}$$

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 that a goal g (e.g. "achieve goal go-home") has three applicable plans *pa*, *pb* and *pc*; that *pa* is selected, and that 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[11].

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

[11] In fact, this is actually an underestimate: it is also possible for the goal to fail because none of the untried relevant plans are applicable in the resulting situation. As noted earlier, we assume in our analysis that goals cannot fail due to there not being applicable plan instances. This is a conservative assumption: relaxing it results in the number of behaviours being even larger.

The number of ways in which a plan can fail is still defined by the same equation — because failure handling happens at the level of goals — but where $n^{\times}(g)$ refers to the new definition:

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

$$n^{\times}(p_0) = \ell \tag{4}$$

$$n^{\times}(p_d) = \ell + \left(n^{\times}(g_d) + \ell\, n^{\checkmark}(g_d)\right) \frac{n^{\checkmark}(g_d)^k - 1}{n^{\checkmark}(g_d) - 1} \tag{5}$$

$$(\text{for } d > 0 \text{ and } n^{\checkmark}(g_d) > 1)$$

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.

Consider a simple case: a depth 1 tree consisting of a goal g (e.g. "achieve goal go-home") with three children: pa, pb, pc. Previously the successful executions corresponded to each of the *pi* (i.e. select a *pi* and execute it). However, with failure handling, we now have the following additional successful executions (as well as additional cases corresponding to different orderings of the plans, e.g. *pb* failing and then *pa* being successfully executed):

- *pa* fails, *pb* is then executed and it succeeds
- *pa* fails, *pb* is then executed and it fails, *pc* is then executed and it succeeds

This leads to a definition of the form

$$n^{\checkmark}(g) = n^{\checkmark}(pa) + n^{\times}(pa)\, n^{\checkmark}(pb) + n^{\times}(pa)\, n^{\times}(pb)\, n^{\checkmark}(pc)$$

We need to account for different orderings of the plans. For instance, the case where the first selected plan succeeds (corresponding to the first term, $n^{\checkmark}(pa)$) in fact applies for each of the j plans, so the first term, including different orderings, is $j\, n^{\checkmark}(p)$.

Similarly, the second term ($n^{\times}(pa)\, n^{\checkmark}(pb)$), corresponding to the case where the initially selected plan fails but the next plan selected succeeds, in fact applies for j initial plans, and then for $j-1$ next plans, yielding $j\,(j-1)\, n^{\times}(p)\, n^{\checkmark}(p)$.

Continuing this process (for $j = 3$) yields the following formulae:

$$n^{\checkmark}(g) = 3\, n^{\checkmark}(p) + 3{\cdot}2\, n^{\times}(p)\, n^{\checkmark}(p) + 3!\, n^{\times}(p)^2\, n^{\checkmark}(p)$$

which generalises to

$$n^{\checkmark}(g) = j\, n^{\checkmark}(p) + j\,(j-1)\, n^{\times}(p)\, n^{\checkmark}(p) + \ldots + j!\, n^{\times}(p)^{j-1}\, n^{\checkmark}(p)$$

resulting in the following equations (again, since failure handling is done at the goal level, the equation for plans is the same as in Section 4.1):

$$n^{\checkmark}(g_d) = \sum_{i=1}^{j} n^{\checkmark}(p_{d-1})\, n^{\times}(p_{d-1})^{i-1}\, \frac{j!}{(j-i)!} \tag{6}$$

$$n^{\checkmark}(p_0) = 1 \tag{7}$$

$$n^{\checkmark}(p_d) = n^{\checkmark}(g_d)^k \ \ (\text{for } d > 0\,) \tag{8}$$

We have used the "standard" BDI failure-handling mechanism of trying alternative applicable plans. Now let us briefly consider an alternative failure-handling mechanism that simply re-posts the event, without tracking which plans have already been attempted. It is

Parameters			Number of				
j	k	d	goals	plans	actions	$n^{\checkmark}(g)$	$n^{\times}(g)$
2	2	3	21	42	62 (13)	128	614
3	3	3	91	273	363 (25)	1,594,323	6,337,425
2	3	4	259	518	776 (79)	1,099,511,627,776	6,523,509,472,174
3	4	3	157	471	627 (41)	10,460,353,203	41,754,963,603

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

Parameters			Number of				
j	k	d	goals	plans	actions	$n^{\checkmark}(g)$	$n^{\times}(g)$
2	2	3	21	42	62 (13)	$\approx 6.33\times10^{12}$	$\approx 1.82\times10^{13}$
3	3	3	91	273	363 (25)	$\approx 1.02\times10^{107}$	$\approx 2.56\times10^{107}$
2	3	4	259	518	776 (79)	$\approx 1.82\times10^{157}$	$\approx 7.23\times10^{157}$
3	4	3	157	471	627 (41)	$\approx 3.13\times10^{184}$	$\approx 7.82\times10^{184}$

Table 2 Illustrative values for $n^{\checkmark}(g)$ and $n^{\times}(g)$ with failure handling

fairly easy to see that this, in fact, creates an *infinite* number of behaviours: suppose that a goal g can be achieved by *pa* or *pb*, then *pa* could be selected, executed resulting in failure, and then *pa* could be selected again, fail again, etc. This suggests that the "standard" BDI failure-handling mechanism is, in fact, more appropriate, in that it avoids an infinite behaviour space, and the possibility of an infinite loop. As discussed earlier (in Section 2), the failure recovery mechanism used by 3APL and 2APL [29] cannot be analysed in a general way, since it depends on the details of the specific agent program; and IRMA [25] does not provide sufficient details to allow for analysis.

Tables 1 and 2 make 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. The number of goals is calculated as follows. At depth 1 there is a single goal (see Figure 4). At depth $n+1$ there are $1+(j\times k\times G(n))$ goals, where $G(n)$ denotes the number of goals in a depth n tree. This gives $G(n) = 1+(j\times k)+(j\times k)^2+\ldots+(j\times k)^{n-1}$. For example, for $j = k = 2$, we have $G(3) = 1+4+16 = 21$. Since each goal has exactly j plans, the number of plans in a tree of depth n is just $j\times G(n)$. We now consider the number of actions. Each non-leaf plan has $\ell\times(k+1)$ actions (since it has k goals, there are $k+1$ places where there are ℓ actions). Each leaf plan has ℓ actions. A tree of depth n has $j\times(j\times k)^{n-1}$ leaf plans. Let $P(n)$ be the number of plans in a depth n tree, which is comprised of $P_n(n)$ non-leaf plans and $P_l(n)$ leaf plans, i.e. $P(n) = P_n(n)+P_l(n)$. Then the number of actions in a depth n tree is $(\ell\times(k+1))\times P_n(n) \;+\; \ell\times P_l(n)$. For example, for $j = k = 2$ and $\ell = 1$, we have that $P(3) = 2\times G(3) = 42$, which is comprised of 32 leaf plans, and 10 non-leaf plans. There are therefore $(1\times3\times10)+(1\times32) = 62$ actions.

4.4 Recurrence Relations

The equations in the previous sections define the functions $n^{\checkmark}$ and $n^{\times}$ as a mutual recurrence on the depth d of a goal-plan tree with a uniform branching structure. The effect of increasing the parameters k and ℓ is evident at each level of the recursion, but it is not so clear what

the effect is of increasing the number of applicable plan instances for any given goal. To better understand this, in this section we relax our uniformity assumption by allowing the number of plans available to vary for goal nodes at different depths in the tree, while still assuming that all nodes at a given depth have the same structure. We will refer to these as *semi-uniform* goal-plan trees. We then derive a set of recurrence relations for $n^{\checkmark}$ and $n^{\times}$ in the presence of failure handling that explicitly show the effect of adding a new plan for a goal at the root of any particular sub-tree.

We begin by defining the generalised notation $n^{\times}(g_{\mathbf{n}})$ and $n^{\checkmark}(g_{\mathbf{n}})$ where $\mathbf{n}$ is a list $(n_d, n_{d-1}, \ldots n_0)$ in which each element n_i represents the number of plans available for goals at depth i of the goal-plan tree. We denote the empty list by $\langle\rangle$ and write $n \cdot \mathbf{n}$ to represent the list with head n and tail $\mathbf{n}$.

We can generalise Equations 3 and 6 to apply to semi-uniform goal-plan trees, as the derivation of these equations depended only on the *sub*-nodes of each goal or plan node having the same structure. This assumption is preserved in this generalised setting. We therefore rewrite these equations below using this new notation, and also express the right hand sides as functions $f^{\times}$ and $f^{\checkmark}$ of $n^{\times}(p_{\mathbf{n}})$ and (for $f^{\checkmark}$) $n^{\checkmark}(p_{\mathbf{n}})$. Our aim is to find a recursive definition of $f^{\times}$ and $f^{\checkmark}$ as a recurrence on n.

$$\begin{aligned} n^{\times}(g_{n\cdot\mathbf{n}}) &= f^{\times}(n, n^{\times}(p_{\mathbf{n}})) \\ n^{\checkmark}(g_{n\cdot\mathbf{n}}) &= f^{\checkmark}(n, n^{\times}(p_{\mathbf{n}}), n^{\checkmark}(p_{\mathbf{n}})) \end{aligned}$$

where

$$\begin{aligned} f^{\times}(n,a) &= n!\,a^n \\ f^{\checkmark}(n,a,b) &= \sum_{i=1}^{n} b\,a^{i-1}\,\frac{n!}{(n-i)!} = \sum_{i=0}^{n-1} \binom{n}{i}\, i!\,a^i\,(n-i)\,b \end{aligned} \tag{9}$$

The equality on the right of the last line above will be useful in the discussion below, and corresponds to the following combinatorial analysis of $f^{\checkmark}$. For a goal $g_{n\cdot\mathbf{n}}$, each successful execution will involve a sequence of i plan executions that fail (for some i, $0 \le i \le n-1$) followed by one plan execution that succeeds. There are $\binom{n}{i}$ ways of choosing the failed plans, which can be ordered in $i!$ ways, and each plan has $a = n^{\times}(p_{\mathbf{n}})$ ways to fail. There are then $n-i$ ways of choosing the final successful plan, which has $b = n^{\checkmark}(p_{\mathbf{n}})$ ways to succeed.

Our goal is now to find an explicit characterisation of the incremental effect of adding an extra plan on $n^{\times}(g_{n\cdot\mathbf{n}})$ and $n^{\checkmark}(g_{n\cdot\mathbf{n}})$ by finding definitions of $f^{\times}$ and $f^{\checkmark}$ as recurrence relations in terms of the parameter n. Deriving the recurrence relation for $f^{\times}$ is straightforward:

$$\begin{aligned} f^{\times}(n,a) &= n!\,a^n \\ &= (n(n-1)\ldots 2{\cdot}1)\ \ (\underbrace{a\,a\ldots a\,a}_{n \text{ times}}) \\ &= (na)\,((n-1)a)\ldots(2a)\,(1a) \end{aligned}$$

This shows that $f^{\times}(0,a) = 1$ and $f^{\times}(n{+}1,a) = (n{+}1)\,a\,f^{\times}(n,a)$

However, the derivation of a recurrence relation for $f^{\checkmark}$ is not as simple. Here we use the technique of first finding an *exponential generating function* (e.g.f.) [31] for the sequence $\{f^{\checkmark}(n,a,b)\}_{n=0}^{\infty}$, and then using that to derive a recurrence relation.

The e.g.f. $F(x)$ of the sequence $\{f^{\checkmark}(n,a,b)\}_{n=0}^{\infty}$ is the function defined by the following power series:

$$F(x) = \sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)\frac{x^n}{n!} \tag{10}$$
$$= \sum_{n=0}^{\infty}\left(\sum_{i=0}^{n-1}\binom{n}{i} i!a^i(n-i)b\right)\frac{x^n}{n!} = \sum_{n=0}^{\infty}\left(\sum_{i=0}^{\infty}\binom{n}{i} i!a^i(n-i)b\right)\frac{x^n}{n!}$$

On the right hand side above we have changed the upper limit of the inner sum to ∞ based on the generalised definition of $\binom{n}{i}$ as $n(n-1)(n-2)...(n-i+1)/i!$, which is valid for all complex numbers n and non-zero integers i [31] and gives $\binom{n}{i} = 0$ for $i > n$.

The right hand side has the form of a product of exponential generating functions [31, Rule $3'$, Section 2.3]:

$$\sum_{n=0}^{\infty}\left(\sum_{i=0}^{\infty}\binom{n}{i}\alpha(i)\beta(n-i)\right)\frac{x^n}{n!} = \left(\sum_{n=0}^{\infty}\alpha(n)\frac{x^n}{n!}\right)\left(\sum_{n=0}^{\infty}\beta(n)\frac{x^n}{n!}\right)$$

where, for our case, $\alpha(n) = n!\,a^n$ and $\beta(n) = n\,b$. Therefore, we can write:

$$F(x) = \left(\sum_{n=0}^{\infty} n!\frac{(ax)^n}{n!}\right)\left(\sum_{n=0}^{\infty} n\,b\,\frac{x^n}{n!}\right)$$

The left hand sum is $G(ax)$ where $G(y) = \sum_n y^n = \frac{1}{1-y}$ [31, Equation 2.5.1]. The right hand sum is equal to $bx\frac{d}{dx}\left(\sum\frac{x^n}{n!}\right)$ [31, Rule $2'$, Section 2.3] $= bx\frac{d}{dx}e^x$ [31, Equation 2.5.3] $= bxe^x$. Thus we have:

$$F(x) = \frac{1}{1-ax}\,bxe^x = \frac{bxe^x}{1-ax}$$

Therefore, $f^{\checkmark}(0,a,b)$ is the constant term in the power series $\sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)\frac{x^n}{n!}$, which is $F(0) = 0$. To find a recurrence relation defining $f^{\checkmark}(n+1,a,b)$ we equate the original definition of $F(x)$ in Equation 10 with our closed form of this function, differentiate each side (to give us a power series with the $f^{\checkmark}(n,a,b)$ values shifted one position to the left), and multiply by the denominator of the closed form, giving us:

$$(1-ax)\frac{d}{dx}\left(\sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)\frac{x^n}{n!}\right) = (1-ax)\frac{d}{dx}\left(\frac{bxe^x}{1-ax}\right)$$
$$\implies (1-ax)\sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)n\frac{x^{n-1}}{n!} = (1-ax)\left(\frac{b(x+1)e^x}{1-ax} + \frac{abxe^x}{(1-ax)^2}\right)$$
$$\implies \sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)n\frac{x^{n-1}}{n!} - \sum_{n=0}^{\infty} a\,f^{\checkmark}(n,a,b)n\frac{x^n}{n!} = b(x+1)e^x + a\frac{bxe^x}{1-ax}$$
$$\implies \sum_{n=0}^{\infty} f^{\checkmark}(n+1,a,b)(n+1)\frac{x^n}{(n+1)!} - \sum_{n=0}^{\infty} an\,f^{\checkmark}(n,a,b)\frac{x^n}{n!}$$
$$= bxe^x + be^x + a\sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)\frac{x^n}{n!}$$
$$= b\sum_{n=0}^{\infty} n\frac{x^n}{n!} + b\sum_{n=0}^{\infty}\frac{x^n}{n!} + a\sum_{n=0}^{\infty} f^{\checkmark}(n,a,b)\frac{x^n}{n!}$$

$$
\begin{aligned}
n^{\checkmark}(g_{n\cdot\mathbf{n}}) &= f^{\checkmark}(n, n^{\times}(p_{\mathbf{n}}), n^{\checkmark}(p_{\mathbf{n}})) \\
n^{\times}(g_{n\cdot\mathbf{n}}) &= f^{\times}(n, n^{\times}(p_{\mathbf{n}}))
\end{aligned}
$$

$$
\begin{aligned}
f^{\checkmark}(0,a,b) &= 0 \\
f^{\checkmark}(n{+}1,a,b) &= (n{+}1)\,(b + a\, f^{\checkmark}(n,a,b)) \\
f^{\times}(0,a) &= 1 \\
f^{\times}(n{+}1,a) &= (n{+}1)\,a\, f^{\times}(n,a)
\end{aligned}
$$

$$
\begin{aligned}
n^{\checkmark}(p_{\langle\rangle}) &= 1 \\
n^{\times}(p_{\langle\rangle}) &= \ell \\
n^{\checkmark}(p_{\mathbf{n}}) &= n^{\checkmark}(g_{\mathbf{n}})^k, \text{ for } \mathbf{n} \neq \langle\rangle \\
n^{\times}(p_{\mathbf{n}}) &= \ell + \left(n^{\times}(g_{\mathbf{n}}) + \ell\, n^{\checkmark}(g_{\mathbf{n}})\right) \frac{n^{\checkmark}(g_{\mathbf{n}})^k - 1}{n^{\checkmark}(g_{\mathbf{n}}) - 1}, \text{ for } \mathbf{n} \neq \langle\rangle
\end{aligned}
$$

Fig. 5 Recurrence relations for the numbers of failures and successes of a goal plan tree in the presence of failure handling

Equating the coefficients of $\frac{x^n}{n!}$ we get:

$$
\begin{aligned}
& f^{\checkmark}(n{+}1,a,b) - an f^{\checkmark}(n,a,b) = bn + b + a f^{\checkmark}(n,a,b) \\
\Longrightarrow\ & f^{\checkmark}(n{+}1,a,b) = b(n{+}1) + a f^{\checkmark}(n,a,b) + an f^{\checkmark}(n,a,b) \\
& \qquad = (n{+}1)(b + a f^{\checkmark}(n,a,b)) \qquad (11)
\end{aligned}
$$

Equation 11 gives us the recurrence relation for the sequence $\{f^{\checkmark}(n,a,b)\}_{n=0}^{\infty}$ that we have been seeking[12]. Figure 5 brings together the equations we have so far for the failure-handling case (including those from the previous section defining $n^{\checkmark}(p_d)$ and $n^{\times}(p_d)$, generalised for semi-uniform trees).

This formulation allows us to more easily see how the behaviour space grows as the number of applicable plans, n, for a goal grows. Considering the meaning of the parameters a and b as the numbers of failures and successes (respectively) of a plan at a level below the current goal node, the equation for $f^{\checkmark}(n{+}1,a,b)$ can be seen to have the following combinatorial interpretation. One plan must be selected to try initially (there are $n{+}1$ choices) and it can either succeed (in one of b different ways), meaning no further plans need to be tried, or fail (in one of a different ways). If it fails, then the goal must then succeed using the remaining n plans, which can occur in $f^{\checkmark}(n,a,b)$ ways.

We can see that the growth in the number of successful executions for a goal grows at a rate greater than $n!a^n$, due to the presence of the b term. The relaxed uniformity constraint used in these recurrence relations also gives us a way to investigate the numbers of traces for goal-plan trees of different semi-uniform shapes. However, in the remainder of this paper we will focus on uniform trees using our original parameter j.

[12] In the simple case when $a = b = 1$ this is listed as sequence A007526 in the On-Line Encyclopedia of Integer Sequences [32]: "the number of permutations of nonempty subsets of $\{1,\cdots,n\}$".

4.5 The Probability of Failing

In Section 4.3 we said that introducing failure handling makes it harder to fail. However, Tables 1 and 2 appear at first glance to contradict this, in that there are many more ways of failing with failure handling than there are without failure handling.

The key to understanding the apparent discrepancy is to consider the *probability* of failing: Tables 1 and 2 merely count 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[13]. 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. $p^{\text{✔}}(g_d) = p^{\text{✔}}(p_{d-1})$. We ignore for the moment the possibility of a goal failing due to there being no applicable plans. This assumption is relaxed later on.

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, i.e. $p^{\text{✘}}(g_d) = p^{\text{✘}}(p_{d-1})^j$. Since failure handling is at the goal level, the equation for plans is unchanged, giving:

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

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

It is not easy to see from the equations what the patterns of probabilities actually are, and so, for illustration purposes, the following table 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 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 both low, and doesn't appear to grow significantly as the goal-plan tree grows.

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%

We now relax the assumption that a goal cannot fail due to plans not being applicable, i.e. that a goal will only fail once all plans have been tried. Unfortunately, relaxing this assumption complicates the analysis. This is because we need to consider the possibility that none of the remaining plans are applicable at *each point* where failure handling attempts to recover.

Let us begin by reconsidering the case where there is no failure handling. We use ε_g to denote the probability of a goal failing due to none of the remaining plans being applicable. We assume, for analysis purposes, that this probability is constant, and in particular, that it does not depend on which plans have already been tried nor on the number of relevant plans remaining.

Then the probability of a goal failing is $p^{\times}(g_d) = \varepsilon_g + (1-\varepsilon_g)\, p^{\times}(p_{d-1})$, i.e. the goal fails either because no plans are applicable or because there are applicable plans and the selected plan fails. As before, the equation for plans is unchanged, since failure handling is done at the goal level. Collecting this gives the following equations for the case without failure handling:

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

Observe that setting $\varepsilon_g = 0$ yields the equations derived earlier, where we assumed that a goal cannot fail due to inapplicable plans.

We now consider the probability of failure *with* failure handling. For a goal with *two* plans we have the following cases:

- The goal can fail due to no plans being applicable (ε_g)
- If there are applicable plans ($(1-\varepsilon_g)\ldots$) then the goal can fail if the first selected plan fails ($p^{\times}(p_{d-1})\ldots$) *and* if failure handling is not successful, which can occur if either

there are no applicable plans (ε_g) or, if there are applicable plans ($(1-\varepsilon_g)\ldots$) and the selected plan fails ($p^{\bigstar}(p_{d-1})$).

Putting this together, for a goal with two plans we have:

$$p^{\bigstar}(g_d) = \varepsilon_g + (1-\varepsilon_g)\, p^{\bigstar}(p_{d-1})\,(\varepsilon_g + (1-\varepsilon_g)\, p^{\bigstar}(p_{d-1}))$$

In the general case of j available plans, we have that a goal can fail if:

A. there are no applicable plans at the outset, with probability ε_g; or
B. there are applicable plans $(1-\varepsilon_g)$, but the selected plan fails ($p^{\bigstar}(p_{d-1})$) and then either there are no further applicable plans (ε_g), or
C. there are applicable plans $(1-\varepsilon_g)$, but the selected plan fails ($p^{\bigstar}(p_{d-1})$) and then either there are no further applicable plans (ε_g),
D. and so on: the reasoning of B is repeated j times.

This gives a definition of the following form:

$$\underbrace{\varepsilon_g}_{\text{A}} + \underbrace{(1-\varepsilon_g)\, p^{\bigstar}(p_{d-1})\,(\varepsilon_g + \underbrace{(1-\varepsilon_g)\, p^{\bigstar}(p_{d-1})\,(\varepsilon_g + \underbrace{\ldots}_{\text{D}})}_{\text{C}})}_{\text{B}}$$

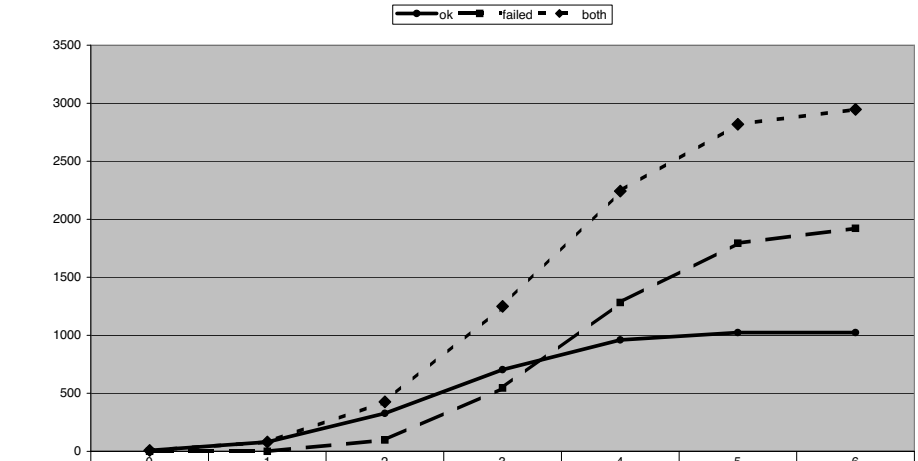
This can be defined in terms of an auxiliary function $p^{\bigstar}(g_d, i)$ which defines the probability of failure for goal g at depth d where there are i remaining relevant plans instances that may (or may not) yield any applicable plan instances:

$$\begin{aligned}
p^{\bigstar}(g_d) &= p^{\bigstar}(g_d, j) \\
p^{\bigstar}(g_d, 1) &= \varepsilon_g + (1-\varepsilon_g)\, p^{\bigstar}(p_{d-1}) \\
p^{\bigstar}(g_d, i+1) &= \varepsilon_g + (1-\varepsilon_g)\, p^{\bigstar}(p_{d-1})\, p^{\bigstar}(g_d, i) \\
p^{\bigstar}(p_0) &= \varepsilon_0 \\
p^{\bigstar}(p_d) &= 1 - [(1-\varepsilon)\,(1-p^{\bigstar}(g_d))^k]
\end{aligned}$$

Observe that setting $\varepsilon_g = 0$ reduces this to the definition derived earlier, since $\varepsilon_g + (1-\varepsilon_g)X$ simplifies to X, and hence $p^{\bigstar}(g_d, i) = p^{\bigstar}(p_{d-1})^i$.

As before, it is not immediately clear from the formulae what the actual patterns of probability are. Considering illustrative examples (see the table below) shows that (a) the overall behaviour is the same as before, and (b) if ε_g is assumed to be relatively low compared with the probability of action failure (ε and ε_0), then it doesn't significantly affect the probabilities.

		No failure handling			With failure handling		
ε_a	d	$\varepsilon_g = 0$	$\varepsilon_g = 0.01$	$\varepsilon_g = 0.05$	$\varepsilon_g = 0$	$\varepsilon_g = 0.01$	$\varepsilon_g = 0.05$
0.05	2	30%	33%	43%	0.64%	2.2%	9.4%
	3	72%	76%	86%	0.81%	2.6%	12.8%
	4	98%	99%	100%	0.86%	2.8%	16.5%
ε_a	d	$\varepsilon_g = 0$	$\varepsilon_g = 0.005$	$\varepsilon_g = 0.01$	$\varepsilon_g = 0$	$\varepsilon_g = 0.005$	$\varepsilon_g = 0.01$
0.01	2	7%	9%	10%	0.006%	0.5%	1.1%
	3	22%	27%	32%	0.006%	0.6%	1.1%
	4	55%	63%	70%	0.006%	0.6%	1.1%



	0	1	2	3	4	5	6
ok	8	80	328	704	960	1024	1024
failed	0	0	98	546	1282	1794	1922
both	8	80	426	1250	2242	2818	2946

failures

Fig. 6 Number of traces (cumulative) for $j = k = 2$, $\ell = 1$, $d = 2$

4.6 Analysis of the Number of Failures

In this section we briefly examine the relationship between the number of action failures and the number of traces.

We have derived equations that calculate the total number of behaviours (with failure handling). But how many of these behaviours involve many action failures? If we make some sort of assumption about the rate of action failure, and hence the number of failures that will occur in an execution of length ℓ, how does this affect the number of behaviours? Do the large numbers that we have seen reduce significantly?

For instance, considering $j = k = 2$, $\ell = 1$ and $d = 2$, there are 1,922 possible executions that result in failure. How many of these involve many failures of actions, and how many involve only a small number of failures? Figure 6 contains (cumulative) numbers which were counted by looking at all possible executions in this (small) case. The x axis shows for a given value $N \in \{0,\ldots,6\}$ how many traces are there that have N or fewer action failures. For instance, for $N = 2$, there are 426 traces that have 2 or fewer action failures. Of these 426 traces, 328 are successful and 98 are unsuccessful.

The question is how to generalise this analysis for larger execution spaces. Clearly, counting all possible executions is not feasible. Instead, we turn to generating functions.

We extend our previous notation by defining $n^{\checkmark}_{af}(s,n)$ and $n^{\times}_{af}(s,n)$ to be the numbers of successful and failed (respectively) executions of a plan body segment s in which exactly n action failures (hence the af subscript) have occurred.

For a given segment s (and particularly for $s = g_d$) , we are interested in computing the sequences $\{n^{\checkmark}_{af}(s,n)\}_{n=0}^{\infty}$ and $\{n^{\times}_{af}(s,n)\}_{n=0}^{\infty}$. This can be achieved by considering the

ordinary (rather than exponential) generating functions [31] for these sequences:

$$F^{\checkmark}_{af}(s,x) = \sum_{n=0}^{\infty} n^{\checkmark}_{af}(s,n)\,x^n$$
$$F^{\times}_{af}(s,x) = \sum_{n=0}^{\infty} n^{\times}_{af}(s,n)\,x^n$$

In contrast to exponential generating functions, the terms in these sums do not include a denominator of $n!$.

An action a has one successful execution, which contains no action failures, so $F^{\checkmark}_{af}(a,x) = 1$ (a power series with the coefficient of x^0 being 1 and all other coefficients being 0). Similarly, $F^{\times}_{af}(a,x) = x$ as there is one failed execution, which has one action failure.

We now consider $F^{\checkmark}_{af}(s_1;s_2)$:

$$\begin{aligned} F^{\checkmark}_{af}(s_1;s_2,x) &= \sum_{n=0}^{\infty} n^{\checkmark}_{af}(s_1;s_2,n)\,x^n \\ &= \sum_{n=0}^{\infty} \left(\sum_{i+j=n} n^{\checkmark}_{af}(s_1,i)\,n^{\checkmark}_{af}(s_2,j) \right) x^n \end{aligned}$$

The inner sum considers all possible ways of allocating the n action failures to the (necessarily) successful and independent (by assumption) executions of s_1 and s_2. The second line above can be rewritten as a product of ordinary generating functions [31, Rule 3, Section 2.2]:

$$\begin{aligned} F^{\checkmark}_{af}(s_1;s_2,x) &= \left(\sum_{n=0}^{\infty} n^{\checkmark}_{af}(s_1,n)\,x^n \right) \left(\sum_{n=0}^{\infty} n^{\checkmark}_{af}(s_2,n)\,x^n \right) \\ &= F^{\checkmark}_{af}(s_1,x)\,F^{\checkmark}_{af}(s_2,x) \end{aligned}$$

Considering $F^{\times}_{af}(s_1;s_2,x)$, we have:

$$\begin{aligned} F^{\times}_{af}(s_1;s_2,x) &= \sum_{n=0}^{\infty} n^{\times}_{af}(s_1;s_2,n)\,x^n \\ &= \sum_{n=0}^{\infty} \left(n^{\times}_{af}(s_1,n) + \sum_{i+j=n} n^{\checkmark}_{af}(s_1,i)\,n^{\times}_{af}(s_2,j) \right) x^n \\ &= \sum_{n=0}^{\infty} n^{\times}_{af}(s_1,n)\,x^n + \sum_{n=0}^{\infty} \left(\sum_{i+j=n} n^{\checkmark}_{af}(s_1,i)\,n^{\times}_{af}(s_2,j) \right) x^n \\ &= F^{\times}_{af}(s_1,x) + \left(\sum_{n=0}^{\infty} n^{\checkmark}_{af}(s_1,n)\,x^n \right) \left(\sum_{n=0}^{\infty} n^{\times}_{af}(s_2,n)\,x^n \right) \\ &= F^{\times}_{af}(s_1,x) + F^{\checkmark}_{af}(s_1,x)\,F^{\times}_{af}(s_2,x) \end{aligned}$$

The second line above is based on the observation that each failed execution of $s_1;s_2$ with n action failures is either a failed execution of s_1 with n action failures occurring in that execution, or is a successful execution of s_1 with i failures followed by a failed execution of s_2 with j failures, where $i+j=n$.

Now, assuming that we know $F^{\checkmark}_{af}(g_d,x)$ and $F^{\times}_{af}(g_d,x)$ for some depth d, we can construct the functions $F^{\checkmark}_{af}(p_d,x)$ and $F^{\times}_{af}(p_d,x)$ by applying the results above to expand the right hand sides of the following equations (which simply replace p_d with its plan body):

$$F^{\checkmark}_{af}(p_d,x) = F^{\checkmark}_{af}(a^{\ell};(g_d;a^{\ell})^k,x)$$
$$F^{\times}_{af}(p_d,x) = F^{\times}_{af}(a^{\ell};(g_d;a^{\ell})^k,x)$$

It remains to define $F^{\checkmark}_{af}(g_d,x)$ and $F^{\checkmark}_{af}(g_d,x)$ in terms of $F^{\checkmark}_{af}(p_{d-1},x)$ and $F^{\checkmark}_{af}(p_{d-1},x)$. To count the successful executions of g_d with n action failures, we must first choose one of the j applicable plans to be the one that ultimately succeeds. We must then choose between 0 and $j-1$ of the remaining applicable plans that were tried but failed, and consider all possible orderings of these plans. The n action failures must be distributed across the failed and successful plans. This leads us to the following derivation of a procedure to construct $F^{\checkmark}_{af}(g_d,x)$:

$$\begin{aligned}
F^{\checkmark}_{af}(g_d,x) &= \sum_{n=0}^{\infty} n^{\checkmark}_{af}(g_d,n)\,x^n \\
&= \sum_{n=0}^{\infty}\left(j\sum_{m=0}^{j-1}\binom{j-1}{m}m!\sum_{i_0+\cdots+i_m=n} n^{\checkmark}_{af}(p_{d-1},i_0)\,n^{\times}_{af}(p_{d-1},i_1)\cdots n^{\times}_{af}(p_{d-1},i_m)\right)x^n \\
&= j\sum_{m=0}^{j-1}\binom{j-1}{m}m!\sum_{n=0}^{\infty}\left(\sum_{i_0+\cdots+i_m=n} n^{\checkmark}_{af}(p_{d-1},i_0)\,n^{\times}_{af}(p_{d-1},i_1)\cdots n^{\times}_{af}(p_{d-1},i_m)\right)x^n \\
&= j\sum_{m=0}^{j-1}\binom{j-1}{m}m!\left(\sum_{n=0}^{\infty} n^{\checkmark}_{af}(p_{d-1},n)\,x^n\right)\left(\sum_{n=0}^{\infty} n^{\times}_{af}(p_{d-1},n)\,x^n\right)^m \\
&= j\sum_{m=0}^{j-1}\binom{j-1}{m}m!\,F^{\checkmark}_{af}(p_{d-1},x)\,F^{\times}_{af}(p_{d-1},x)^m
\end{aligned}$$

Constructing $F^{\times}_{af}(g_d,x)$ is simpler. A failed execution of a goal involves failed attempts to execute all j applicable plans. All $j!$ orderings of these plans must be considered. This gives us the following construction for $F^{\times}_{af}(g_d,x)$:

$$\begin{aligned}
F^{\times}_{af}(g_d,x) &= \sum_{n=0}^{\infty} n^{\times}_{af}(g_d,n)\,x^n \\
&= \sum_{n=0}^{\infty}\left(j!\sum_{i_1+\cdots+i_j=n} n^{\times}_{af}(p_{d-1},i_1)\cdots n^{\times}_{af}(p_{d-1},i_j)\right)x^n \\
&= j!\left(\sum_{n=0}^{\infty} n^{\times}_{af}(p_{d-1},n)\,x^n\right)^j \\
&= j!\,F^{\times}_{af}(p_{d-1},x)^j
\end{aligned}$$

The equations above define a recursive procedure for computing $F^{\checkmark}_{af}(g_d,x)$ and $F^{\times}_{af}(g_d,x)$ for a given d. Furthermore, given any ordinary generating function $F(x)$ that generates a sequence a_n, the sequence of partial sums of a_n is generated by $F(x)/(1-x)$ [31, Exercise 32, Chapter 2]. Therefore we can generate the numbers of successful and failed executions of g_d with *at most* n action failures using the functions $F^{\checkmark}_{af}(g_d,x)/(1-x)$ and $F^{\times}_{af}(g_d,x)/(1-x)$.

The Sage mathematical software system[14] was used to obtain the power series representations of these functions[15] for the values of d, j, k and l shown in Figures 7 and 8[16].

Examining Figures 7 and 8 we can conclude two things. On the one hand, the number of traces really explodes for larger numbers of action failures. For example, in Figure 7 most traces have between 10 and 16 action failures. On the other hand, although making assumptions about the number of failures that can occur does reduce the number of possible traces, the number of traces is still quite large (note the scale on the y-axis). For instance, in Figure 7, even if we limit executions to at most 5 action failures, there are still 1,186,693,266 possible traces. Similar conclusions can be drawn from Figure 8 where, even assuming at most 10 action failures, there are still *many many*[17] traces.

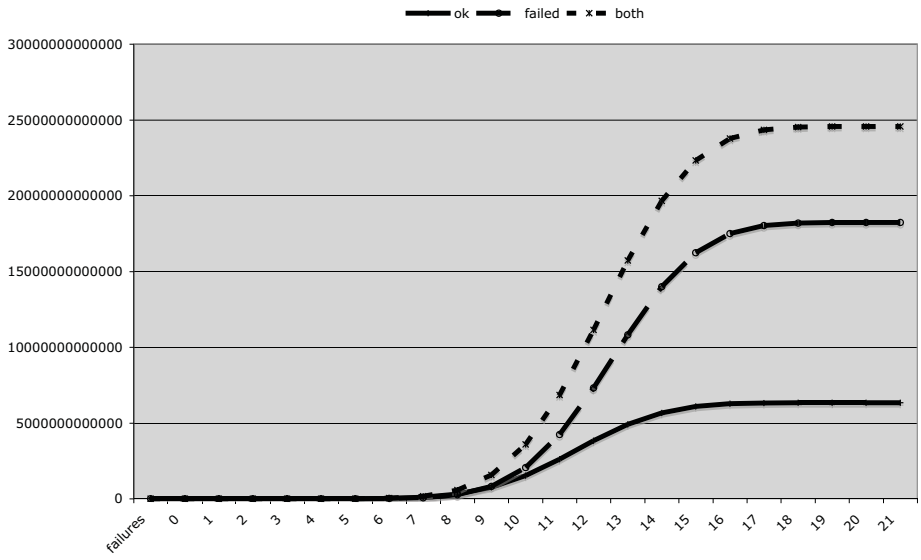


Fig. 7 Number of traces (cumulative) for $j = k = 2$, $\ell = 1$, $d = 3$

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.

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

[15] There are a finite number of actions that can be attempted during any execution of a goal-plan tree, and this defines a bound on the total number of action failures that can occur. Thus $P^{\checkmark}_{af}(g_d, x)$ and $F^{\times}_{af}(g_d, x)$ are polynomial functions of finite order. The functions generating the partial sums have infinite power series, but only a finite number of terms need to be generated as the coefficients increase to a maximum value, which is then repeated infinitely.

[16] The Sage script used to compute this can be found at http://www.sagenb.org/home/pub/639

[17] 4,417,766,935,416,766,347,021,913,820,447,697,963 to be precise.

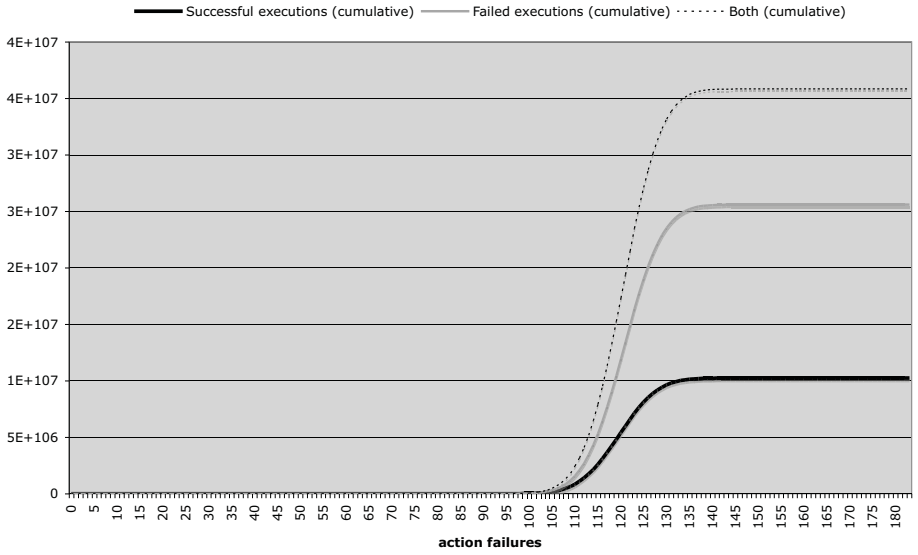


Fig. 8 Number of traces (cumulative) for $j = k = d = 3$ and $\ell = 1$

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 results apply to real systems we:

1. Compare our abstract BDI execution model with results from a real BDI platform (namely JACK [16]). 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.
2. We analyse 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 (and semi-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 Tables 1 and 2 apply to real applications?

5.1 A Real Platform

In order to compare a real BDI platform's execution with the results of our abstract BDI execution model we implemented the two goal-plan trees in the appendix in the JACK agent programming language[18]. The structure of the plans and events[19] precisely mirrors the structure of the tree. As in the goal-plan tree, each event has two relevant plans, both of which are always applicable, and selectable in either order. Actions were implemented using code that printed out the action name, and then, depending on a condition (described below) either continued execution, or triggered failure (and printed out a failure indicator):

[18] The code is available upon request from the authors.

[19] JACK models a goal as a "`BDIGoalEvent`".

```
System.out.print("a"); // Action "a"
if ((N.i & 1)==0) {
  System.out.print("x");
  false; // trigger failure
}
```

The conditions that determined whether an action failed or succeeded, and which plan was selected first, were controlled by an input (`N.i`, a Java class variable). A test harness systematically generated all inputs, thus forcing all decision options to be explored.

The results matched those computed by the Prolog code of Figure 3, giving precisely the same six traces for the smaller tree, and the same 162 traces for the larger tree. This indicates that our abstract BDI execution model is indeed an accurate description of what takes place in a real BDI platform (specifically JACK).

Note that we selected JACK for two reasons, one is that it is a modern, well-known, industry-strength BDI platform. The other, more important, reason, is that JACK is a descendent of a line of BDI platforms going back to PRS, and thus is a good representative for a larger family of BDI platforms. In other words, by showing that the BDI execution model analysed matches JACK's model, we are also able to argue that it matches the execution of JACK's predecessors (including PRS and dMARS), and close relatives (e.g. UM-PRS and JAM).

5.2 A Real Application

We now consider to what extent real systems have deep and branching goal-plan trees, and to what extent the large numbers shown in Tables 1 and 2 apply to real applications, rather than to uniform goal-plan trees. As an example of a real application we consider a recent industrial application at Daimler which used BDI agents to realise agile business processes [33]. Note that finding a suitable application is somewhat challenging: we need an application that is real (not a toy system). However, in order to be able to analyse it, the application has to be BDI-based, and furthermore, details about the application's goal-plan tree need to be available. Unfortunately, many of the reported BDI-based industrial applications do not provide sufficient details about their internals to allow analysis to be carried out.

Figure 9 shows[20] a goal-plan tree from [33] which has "*60 achieve goals in up to 7 levels. 10 maintain goals, 85 plans and about 100 context variables*" [33, Page 41]. Unlike the typical goal-plan trees used in BDI platforms, the tree in Figure 9 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 combinations of ℓ actions, whereas it is clear from [33] that their plans are in fact more complicated, and can contain nested decision making (for example see [33, 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 9 has size in between the first two rows of Table 3. Comparing the number

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

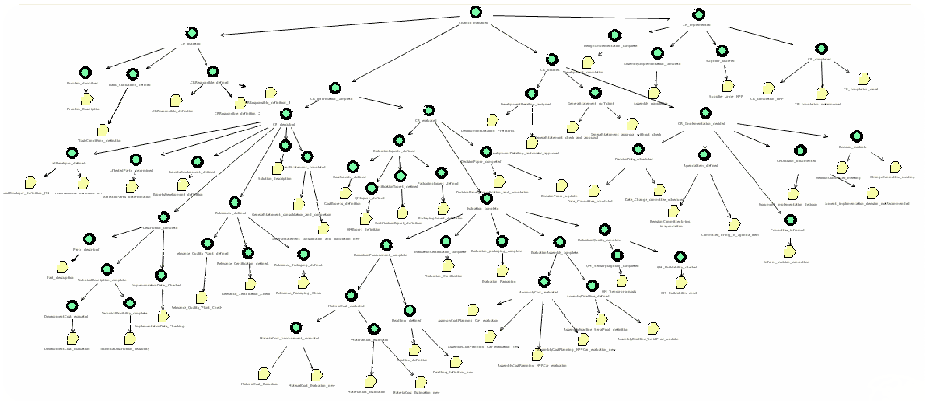


Fig. 9 Goal-plan tree from [33, Figure 6] (reproduced with permission from IFAAMAS)

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 9 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.

Parameters			Number of		No 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 9 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 $n^{\times}(g)$

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, for a uniform goal-plan tree with depth 3 and $j = k = 2$ adding failure handling 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? Before we can answer this question, we need to consider what is being tested. Testing is typically carried out at the levels of individual components (unit testing), collections of components (integration testing), and the system as a whole.

Consider testing of a whole system. The behaviour space sizes depicted in Tables 1, 2 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. The reason for this is that (as discussed in Section 1), a test corresponds to checking one execution trace, and our overall confidence of the correctness of the system is roughly correlated with the proportion of possible traces that have been tested (and found to be correct). If we run, say, 1,000 tests, then our confidence of correctness will (all else being equal) be higher if the system has 100,000 possible traces than if it has 100,000,000 possible traces.

In fact, this situation is even worse when we consider not only the *number* of possible executions but also the *probability* of failing: 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". Furthermore, as shown in Section 4.6, although making assumptions about the possible numbers of action failures that can occur in a given execution reduces the number of possible behaviours, there are still many many behaviours, even for relatively small trees (e.g. $j = k = d = 3$).

So system testing of BDI agents seems to be impractical. What about unit testing and integration testing? Although unit and integration testing are useful, it is not always clear how to apply them usefully to agent systems where the interesting behaviour is complex and possibly emergent. For example, given an ant colony optimisation system [34], testing a single ant doesn't provide much useful information about the correct functioning of the whole system. Similarly, for BDI agents, when testing a sub-goal it can be difficult to ensure that testing covers all the situations in which the goal may be attempted. Likewise, when testing an agent without the rest of the system (including other agents that it interacts with) it can be hard to ensure adequate coverage of different possibilities.

We do need to 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, even though smaller, it is still quite large, and this did cause problems in validation[21]:

> "*One of the big challenges during the test phase was to keep the model consistent and to define the right context conditions that result in the correct execution for all scenarios. Therefore more support for dependency analysis, automated simulation and testing of the process models is needed*" [33, p42].

[21] Burmeister *et al.* also noted that "With this approach changes in the process can be quickly modeled and tested. Thus errors in the models can be discovered and corrected in a short time". They were discussing the advantages of executable models, and arguing that being able to execute the model allowed for testing, which was useful in detecting errors in the model. While being able to execute a model is undoubtedly useful, there is no evidence given (nor is a specific claim made) that testing is sufficient for assuring the correctness of an agent system.

So where does that leave us with respect to testing agent systems? The conclusion seems to be that testing a whole BDI system is not feasible. There are a number of recommendations that could be drawn:

- **Keep BDI goal-plan trees shallow and sparse:** which keeps the number of behaviours small. The issue with this approach is that we lose the benefits of the BDI approach: a reasonably large number of behaviours is desirable in that it provides flexibility and robustness.
- **Avoid failure handling:** since failure handling is a large contributor to the behaviour space, we could modify agent languages to disable failure handling. Again, this is not a useful approach because disabling failure handling removes the benefits of the approach, specifically the ability to recover from failures.
- **Make testing more sophisticated:** could testing coverage perhaps be improved by incorporating additional information such as domain knowledge, and a detailed model of the environment (which indicates the possible failure modes and their probabilities)? The answer is not known, but this is a potentially interesting area for further work. However, the sheer size of the behaviour space does not encourage much optimism for this approach.
  Another, related, direction is to see whether regular patterns exist in the behaviour space. Since the failure mechanism has a certain structure, it may be that this results in a behaviour space that is large, but, in some sense, structured. If such structure exists, it may be useful it making agents more testable. However, at this point in time, this is a research direction that may or may not turn out to be fruitful; not a viable testing strategy.
- **Supplement testing with alternative means of assurance:** since testing is not able to cover a large behaviour space, we should consider other forms of assurance. A promising candidate here is some form of formal methods. Unfortunately, formal methods techniques are not yet applicable to industry-sized agent systems (we return to this below, under “Future Work”).
- **Proceed with caution:** accept that BDI agent systems are in general robust (due to their failure-handling mechanisms), but that there is, at present, no practical way of assuring that they will be behave appropriately in all possible situations. It is worth noting that humans are similar in this respect: whilst we can train, examine and certify a human for a certain role (e.g. a pilot or surgeon), there is no way of assuring that they will behave appropriately in all situations, and so in situations where incorrect behaviour may have dire consequences, the surrounding system needs to have safety precautions built in (e.g. a process that double-checks information, or a backup system such as a co-pilot).

Future Work

There is room for extending the analysis of Section 4. Firstly, our analysis is for a *single* goal within a *single* agent. Multiple agents that are collaborating to achieve a single high-level goal can be viewed as having a shared goal-plan tree where certain goals and/or plans are allocated to certain agents. Of course, in such a “distributed goal plan tree” there is concurrency. Once concurrency is introduced, it would be useful to consider whether certain interleavings of concurrent goals are in fact equivalent. Furthermore, we have only considered achievement goals. It would be interesting to consider other types of goals [35]. Secondly, our analysis has focused on BDI agents, which are just one particular type of agent. It would be interesting to consider other sorts of agent systems, and, more broadly, other sorts of adaptive systems.

Another area for refinement of the analysis is to make it less abstract. Two specific areas where it could be made more detailed are resources and the environment. Our analysis does not consider resources or the environment directly, instead, it considers that actions may fail for a range of reasons which might include resource issues, or environmental issues. The analysis could be extended to explicitly consider resources and their interaction with goals (e.g. [36]). It could also be extended with an explicit model of the environment.

Whilst our analysis did consider a real application, 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. A key challenge is finding suitable applications that are BDI-based; are sufficiently complex (ideally a real application); and have detailed design information available (and preferably source code). Another challenge is the methodology: we analysed the "shape" of the goal-plan tree of the Daimler workflow application, but did not have access to run the system. An alternative methodology, which requires access to the implemented system and probably the source code, is to run it, and force it to generate all traces for sub-goals[22] (which would require modification of either the source code or the underlying agent platform). Once we have collected data on the shape of real-world industrial applications, we will be able to analyse whether uniform and semi-uniform goal-plan trees are good models of these types of system, or whether we should seek ways to further relax our uniformity assumption.

More importantly, having highlighted the infeasibility of verifying BDI agent systems through testing, we need to find other ways of verifying such systems.

An approach that has some promise is the automatic generation of test cases for agent systems [37, 6]. However, the size of the behaviour space suggests that the number of test cases needed may be very large, and that testing for failed plan execution is difficult. One interesting, and potentially promising, avenue is to use formal techniques to help guide the test generation process (e.g. symbolic execution or specification-guided testing) [38].

Another promising approach that has attracted some interest is model checking[23] of agent systems [39–41]. This work is promising because model checking techniques use a range of abstractions to cover a large search space without having to deal with individual cases one-at-a-time (e.g. [42, 43]). Furthermore, because verifying a sub-goal considers all possibilities, it is possible to combine the verification of different sub-goals. However, more work is needed: Raimondi and Lomuscio [41] verify systems where agents are defined abstractly, i.e. not in terms of plans and goals; the MABLE agent programming language [39] is actually an imperative language augmented with certain agent features, not a BDI language; and the work of Bordini *et al.* [40] does not include failure handling. In general, the state of the art in model checking agent system implementations is still limited to quite small systems.

Acknowledgements We would like to thank members of the Department of Information Science at the University of Otago for discussions relating to this paper. We would also like to thank Lin Padgham for her comments on a draft of this paper. Much of the work on this paper was done while Winikoff was on sabbatical from RMIT, visiting the University of Otago.

[22] Generating all traces of the top level goal is not likely to be feasible.

[23] There has also been work on deductive verification, but (based on research into the verification of concurrent systems) this appears to be less likely to result in verification tools that are both (relatively) easy to use and applicable to real systems.

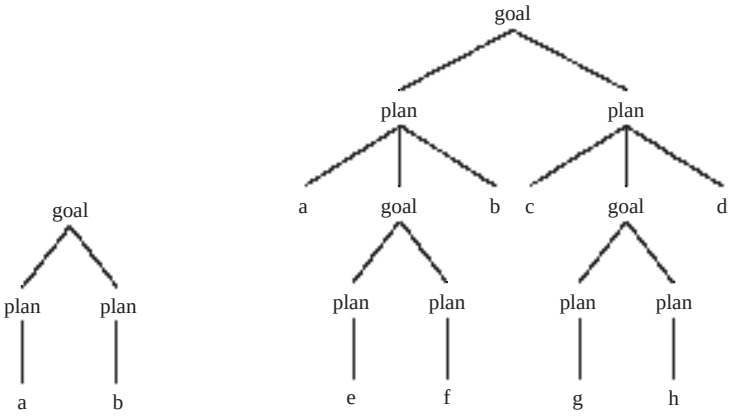
References

1. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Chichester, England (2002). ISBN 0 47149691X
2. Munroe, S., Miller, T., Belecheanu, R., Pechoucek, 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)
3. 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)
4. 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)
5. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
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). ISBN 0-470-86120-7
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. Erol, K., Hendler, J.A., Nau, D.S.: Htn planning: Complexity and expressivity. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI), pp. 1123–1128. AAAI Press (1994)
13. de Silva, L., Padgham, L.: A comparison of BDI based real-time reasoning and HTN based planning. In: G. Webb, X. Yu (eds.) AI 2004: Advances in Artificial Intelligence, *Lecture Notes in Computer Science*, vol. 3339, pp. 1167–1173. Springer, Berlin/Heidelberg (2004)
14. Erol, K., Hendler, J., Nau, D.: Complexity results for htn planning. Annals of Mathematics and Artificial Intelligence **18**(1), 69–93 (1996)
15. Paolucci, M., Shehory, O., Sycara, K.P., Kalp, D., Pannu, A.: A planning component for RETSINA agents. In: N.R. Jennings, Y. Lespérance (eds.) Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL '99, Orlando, Florida, USA, July 15-17, 1999, Proceedings, *Lecture Notes in Computer Science*, vol. 1757, pp. 147–161. Springer, Berlin/Heidelberg (2000)
16. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News (2) (1999). URL http://www.agentlink.org/newsletter/2/newsletter2.pdf
17. Huber, M.J.: JAM: A BDI-theoretic mobile agent architecture. In: Proceedings of the Third International Conference on Autonomous Agents (Agents'99), pp. 236–243. ACM Press (1999)
18. d'Inverno, M., Kinny, D., Luck, M., Wooldridge, M.: A formal specification of dMARS. In: M. Singh, A. Rao, M. Wooldridge (eds.) Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages, *Lecture Notes in Artificial Intelligence*, vol. 1365, pp. 155–176. Springer, Berlin/Heidelberg (1998)
19. Georgeff, M.P., Lansky, A.L.: Procedural knowledge. Proceedings of the IEEE, Special Issue on Knowledge Representation **74**(10), 1383–1398 (1986)
20. Ingrand, F.F., Georgeff, M.P., Rao, A.S.: An architecture for real-time reasoning and system control. IEEE Expert **7**(6), 33–44 (1992)
21. Lee, J., Huber, M.J., Kenny, P.G., Durfee, E.H.: UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In: Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94), pp. 842–849 (1994)

22. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007). ISBN 0470029005
23. Morley, D., Myers, K.: The SPARK agent framework. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 714–721. IEEE Computer Society, Washington, DC, USA (2004)
24. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: R.H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 149–174. Springer (2005)
25. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Computational Intelligence **4**, 349–355 (1988)
26. 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)
27. 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)
28. Georgeff, M.: Service orchestration: The next big challenge. DM Review Special Report (2006). URL http://www.dmreview.com/specialreports/20060613/1056195-1.html. (2006)
29. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3), 214–248 (2008)
30. 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)
31. Wilf, H.S.: generatingfunctionology, second edn. Academic Press Inc., Boston, MA (1994). URL http://www.math.upenn.edu/~wilf/gfology2.pdf
32. Sloane, N.J.A.: The on-line encyclopedia of integer sequences. http://www.research.att.com/~njas/sequences/ (2007)
33. 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)
34. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004). ISBN 0-262-04219-3
35. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: A unifying framework. In: Proceedings of the Seventh Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 713–720. IFAAMAS (2008)
36. Thangarajah, J., Winikoff, M., Padgham, L., Fischer, K.: Avoiding resource conflicts in intelligent agents. In: F. van Harmelen (ed.) Proceedings of the 15th European Conference on Artificial Intelligence (ECAI), pp. 18–22. IOS Press (2002)
37. Nguyen, C.D., Perinirini, A., Tonella, P.: Automated continuous testing of multi-agent systems. In: Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS) (2007)
38. Dwyer, M.B., Hatcliff, J., Pasareanu, C., Robby, Visser, W.: Formal software analysis: Emerging trends in software model checking. In: Future of Software Engineering 2007, pp. 120–136. IEEE Computer Society, Los Alamitos, CA (2007)
39. Wooldridge, M., Fisher, M., Huget, M.P., Parsons, S.: Model checking multi-agent systems with MABLE. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 952–959. ACM Press (2002)
40. 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)
41. Raimondi, F., Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. J. Applied Logic **5**(2), 235–251 (2007)
42. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992)
43. Fix, L., Grumberg, O., Heyman, A., Heyman, T., Schuster, A.: Verifying very large industrial circuits using 100 processes and beyond. In: D. Peled, Y.K. Tsay (eds.) Automated Technology for Verification and Analysis, *Lecture Notes in Computer Science*, vol. 3707, pp. 11–25. Springer, Berlin/Heidelberg (2005)

A Example Goal-Plan Trees and their Expansions

Suppose we have the following two trees: `sample` (left) and `sample2` (right). The trees correspond to $j = 2$, $k = \ell = 1$, $d = 1$ for `sample` and $d = 2$ for `sample2`.



Then these trees can be expanded respectively into the following sequences of actions, where a letter indicates the execution of an action, and a ✘ indicates failure. As predicted by our formulae, there are 4 successful executions and 2 unsuccessful executions for the first tree:

a	a ✘ b	a ✘ b ✘
b	b ✘ a	b ✘ a ✘

For the second tree, the expansions are the following 162 possibilities (consisting of 64 successful, and 98 unsuccessful traces).

a e b	a f b ✘ c h d	c g d ✘ a e b	c h d ✘ a f b ✘
a e b ✘ c g d	a f b ✘ c h d ✘	c g d ✘ a e b ✘	c h d ✘ a f ✘ e b
a e b ✘ c g d ✘	a f b ✘ c h ✘ g d	c g d ✘ a e ✘ f b	c h d ✘ a f ✘ e b ✘
a e b ✘ c g ✘ h d	a f b ✘ c h ✘ g d ✘	c g d ✘ a e ✘ f b ✘	c h d ✘ a f ✘ e ✘
a e b ✘ c g ✘ h d ✘	a f b ✘ c h ✘ g ✘	c g d ✘ a e ✘ f ✘	c h d ✘ a ✘
a e b ✘ c g ✘ h ✘	a f b ✘ c ✘	c g d ✘ a f b	c h ✘ g d
a e b ✘ c h d	a f ✘ e b	c g d ✘ a f b ✘	c h ✘ g d ✘ a e b
a e b ✘ c h d ✘	a f ✘ e b ✘ c g d	c g d ✘ a f ✘ e b	c h ✘ g d ✘ a e b ✘
a e b ✘ c h ✘ g d	a f ✘ e b ✘ c g d ✘	c g d ✘ a f ✘ e b ✘	c h ✘ g d ✘ a e ✘ f b
a e b ✘ c h ✘ g d ✘	a f ✘ e b ✘ c g ✘ h d	c g d ✘ a f ✘ e ✘	c h ✘ g d ✘ a e ✘ f b ✘
a e b ✘ c h ✘ g ✘	a f ✘ e b ✘ c g ✘ h d ✘	c g d ✘ a ✘	c h ✘ g d ✘ a e ✘ f ✘
a e b ✘ c ✘	a f ✘ e b ✘ c g ✘ h ✘	c g ✘ h d	c h ✘ g d ✘ a f b
a e ✘ f b	a f ✘ e b ✘ c h d	c g ✘ h d ✘ a e b	c h ✘ g d ✘ a f b ✘
a e ✘ f b ✘ c g d	a f ✘ e b ✘ c h d ✘	c g ✘ h d ✘ a e b ✘	c h ✘ g d ✘ a f ✘ e b
a e ✘ f b ✘ c g d ✘	a f ✘ e b ✘ c h ✘ g d	c g ✘ h d ✘ a e ✘ f b	c h ✘ g d ✘ a f ✘ e b ✘
a e ✘ f b ✘ c g ✘ h d	a f ✘ e b ✘ c h ✘ g d ✘	c g ✘ h d ✘ a e ✘ f b ✘	c h ✘ g d ✘ a f ✘ e ✘
a e ✘ f b ✘ c g ✘ h d ✘	a f ✘ e b ✘ c h ✘ g ✘	c g ✘ h d ✘ a e ✘ f ✘	c h ✘ g d ✘ a ✘
a e ✘ f b ✘ c g ✘ h ✘	a f ✘ e b ✘ c ✘	c g ✘ h d ✘ a f b	c h ✘ g ✘ a e b
a e ✘ f b ✘ c h d	a f ✘ e ✘ c g d	c g ✘ h d ✘ a f b ✘	c h ✘ g ✘ a e b ✘
a e ✘ f b ✘ c h d ✘	a f ✘ e ✘ c g d ✘	c g ✘ h d ✘ a f ✘ e b	c h ✘ g ✘ a e ✘ f b
a e ✘ f b ✘ c h ✘ g d	a f ✘ e ✘ c g ✘ h d	c g ✘ h d ✘ a f ✘ e b ✘	c h ✘ g ✘ a e ✘ f b ✘
a e ✘ f b ✘ c h ✘ g d ✘	a f ✘ e ✘ c g ✘ h d ✘	c g ✘ h d ✘ a f ✘ e ✘	c h ✘ g ✘ a e ✘ f ✘
a e ✘ f b ✘ c h ✘ g ✘	a f ✘ e ✘ c g ✘ h ✘	c g ✘ h d ✘ a ✘	c h ✘ g ✘ a f b
a e ✘ f b ✘ c ✘	a f ✘ e ✘ c h d	c g ✘ h ✘ a e b	c h ✘ g ✘ a f b ✘
a e ✘ f ✘ c g d	a f ✘ e ✘ c h d ✘	c g ✘ h ✘ a e b ✘	c h ✘ g ✘ a f ✘ e b
a e ✘ f ✘ c g d ✘	a f ✘ e ✘ c h ✘ g d	c g ✘ h ✘ a e ✘ f b	c h ✘ g ✘ a f ✘ e b ✘
a e ✘ f ✘ c g ✘ h d	a f ✘ e ✘ c h ✘ g d ✘	c g ✘ h ✘ a e ✘ f b ✘	c h ✘ g ✘ a f ✘ e ✘
a e ✘ f ✘ c g ✘ h d ✘	a f ✘ e ✘ c h ✘ g ✘	c g ✘ h ✘ a e ✘ f ✘	c h ✘ g ✘ a ✘
a e ✘ f ✘ c g ✘ h ✘	a f ✘ e ✘ c ✘	c g ✘ h ✘ a f b	c ✘ a e b

ae✘f✘chd
ae✘f✘chd✘
ae✘f✘ch✘gd
ae✘f✘ch✘gd✘
ae✘f✘ch✘g✘
ae✘f✘c✘
afb
afb✘cgd
afb✘cgd✘
afb✘cg✘hd
afb✘cg✘hd✘
afb✘cg✘h✘

a✘cgd
a✘cgd✘
a✘cg✘hd
a✘cg✘hd✘
a✘cg✘h✘
a✘chd
a✘chd✘
a✘ch✘gd
a✘ch✘gd✘
a✘ch✘g✘
a✘c✘
cgd

cg✘h✘afb✘
cg✘h✘af✘eb
cg✘h✘af✘eb✘
cg✘h✘af✘e✘
cg✘h✘a✘
chd
chd✘aeb
chd✘aeb✘
chd✘ae✘fb
chd✘ae✘fb✘
chd✘ae✘f✘
chd✘afb

c✘aeb✘
c✘ae✘fb
c✘ae✘fb✘
c✘ae✘f✘
c✘afb
c✘afb✘
c✘af✘eb
c✘af✘eb✘
c✘af✘e✘
c✘a✘