Journal of Artificial Intelligence Research 51 (2014) 71-131 Submitted 5/14; published 9/14

On the Testability of BDI Agent Systems

Michael Winikoff MICHAEL.WINIKOFF@OTAGO.AC.NZ
Stephen Cranefield STEPHEN.CRANEFIELD@OTAGO.AC.NZ
Department of Information Science
University of Otago
New Zealand

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.

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* (Wooldridge, 2002). Agent-based systems have been increasingly finding deployment in a wide range of applications (e.g. Munroe, Miller, Belecheanu, Pechoucek, McBurney, & Luck, 2006; Benfield, Hendrickson, & Galanti, 2006).

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[1]. However, it is generally accepted that adaptive systems can exhibit a wide and complex range of behaviours, making testing hard. For example:

> Validation through extensive tests was mandatory .... However, the task proved challenging .... Agent-based systems explore realms of behaviour outside people's expectations and often yield surprises. (Munroe et al., 2006, Section 3.7.2)

That is, there is an intuition that agent systems exhibit complex behaviour, which makes them hard to test. In this paper we explore this intuition, focusing on the well known Belief-Desire-Intention (BDI) approach to realising adaptive and flexible agents (Rao & Georgeff,

1. Although there is considerable research on formal methods in the context of agent systems (Dastani, Hindriks, & Meyer, 2010), it is not yet ready for real world application (see Section 7), and there are concerns about the scope of the work and its applicability (Winikoff, 2010).

1991; Bratman, 1987), which has been demonstrated to be practically applicable, resulting in reduced development cost and increased flexibility (Benfield et al., 2006).

We explore the intuition that "agent systems are hard to test" by analysing both the space of possible behaviours of BDI agents, that is, the number of paths through a BDI program, and the probability of failure. We focus on BDI agents because they provide a well-defined execution mechanism that can be analysed, and 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).

We derive the number of paths through a BDI program as a function of various parameters (e.g. the number of applicable plans per goal and the failure fate). This naturally leads us also to consider how the number of paths is affected by these various parameters. As might be expected, we show that the intuition that "agent systems are hard to test" is correct, i.e. that agent systems have a very large number of paths. We also show that BDI agents are *harder* to test than procedural programs, by showing that the number of paths through a BDI program is much larger than the number of paths through a similarly-sized procedural program.

The contribution of this paper is threefold. Firstly, it confirms the intuition that BDI programs are hard to test. Secondly, it does so by *quantifying* the number of paths, as a function of parameters of the BDI program. Thirdly, we find some surprising results about how the parameters influence the number of paths.

Although there has recently been increasing interest in testing agent systems (Zhang, Thangarajah, & Padgham, 2009; Ekinci, Tiryaki, Çetin, & Dikenelli, 2009; Gomez-Sanz, Botía, Serrano, & Pavón, 2009; Nguyen, Perini, & Tonella, 2009b), there has been surprisingly little work on determining the feasibility of testing agent systems in the first place. Padgham and Winikoff (2004, pp. 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, Farwer and Bordini (2008) 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.

We now briefly address a number of possible criticisms of this work, by considering existing work.

1. Is the number of paths a useful metric for assessing testability?
   We consider the related area of software testing (Section 1.1) and argue that the metric is a well established one, and is appropriate to use to assess testability.

2. Isn't this just an obvious corollary of the complexity of HTN planning?
   We consider in detail the HTN planning problem (Section 1.2) and argue that although the BDI execution cycle has certain similarities with HTN planning, the differences are significant, and, in particular, they mean that the problem of HTN planning is simply different from the problem of testing BDI programs.

3. Why use combinatorial analysis, rather than complexity analysis?
   Our combinatorial analysis is more precise: it yields formulae for the exact number

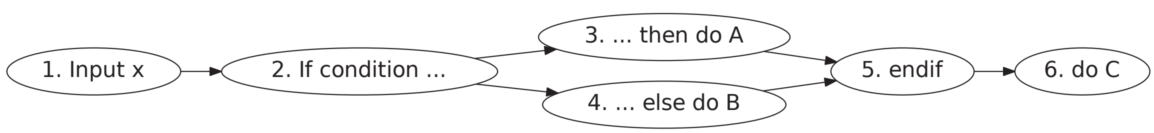
of paths, and the exact probabilities of failure. The latter (see Section 4.5) is more informative than just having an order of magnitude complexity. Additionally it allows us to consider issues that complexity analysis would not address, such as the effect of number of failures on the number of paths.

1.1 Software Testing

We are trying to assess how hard agent systems are to test. More concretely, given a BDI agent program, we want to know how hard that program is to test. This can be reduced directly to the question of test set adequacy. An agent program P is easy to test precisely to the extent that there exists a test set T which is adequate for testing P, where T is not infeasibly large. Conversely, an agent program P is hard to test to the extent that an adequate test set T would have to be infeasibly large. In other words, the "hardness" of testing a program is directly assessed by the size required for a test set to be adequate with respect to a suitable adequacy criteria.

There are many criteria that can be used to assess whether a given set of tests is adequate (for a recent overview, see Mathur, 2008). Given that we are interested in assessing the difficulty of testing a given program, we are clearly looking at "white box" testing. Furthermore, we will be working with abstract "goal-plan trees" rather than detailed programs (see Section 2). This means that we need to consider control-flow based metrics, rather than data-flow, since an abstract goal-plan tree does not contain data-flow information.

Focussing on white box testing criteria that are control-flow based, a basic and very long-standing criterion for assessing test set adequacy is that all paths in the program be covered (Miller & Maloney, 1963). For example, consider a program of the following form.



There are two paths through this program: (1, 2, 3, 5, 6) and (1, 2, 4, 5, 6), and an adequate test set must have at least two tests to be adequate: one to exercise the first path, and another to exercise the second. In this case a test set with only a single test will be inadequate, and will result in part of the program not being executed at all during testing.

An obvious complication in covering all paths in a program is that any loop will result in an infinite number of paths, since the loop can potentially be executed any number of times. The standard technique for dealing with this is to bound the length of paths, or the number of executions of a loop (Zhu, Hall, & May, 1997, p. 375). Bounding the execution of loops can be done either by calculating an upper bound on the number of iterations based on the data (Mathur, 2008, p. 53), or by only considering paths in which loops are executed zero times or one time (Mathur, 2008, p. 408).

One question that might be asked is why we consider all paths, rather than a weaker criterion. Agent applications typically involve environments that are non-episodic. That is, the environment's history matters. This means that the behaviour of a given plan or goal is, in general, sensitive to the agent's history, and hence we need to consider the different possible histories. Achieving a goal may be different if it is being done as the first thing the agent does, or after a failed plan which has already performed a number of actions. This means that it makes sense to consider a path-based criterion for testing.

Furthermore, although the "all paths" adequacy criterion is often considered to be impractical, the reason appears to be primarily the existence of an infinite number of paths in the presence of loops. For instance, Zhu et al. (1997, p. 375) say the plan coverage criterion "is too strong to be practically useful for most programs, because there can be an infinite number of different paths in a program with loops". In our setting, where we do not have loops, the existence of an infinite number of paths is not an issue, so considering the number of paths is possible.

We therefore use the number of paths as our proxy measure for testing difficulty: if there are few paths through the program, then an adequate test set (according to the "all paths" criterion) will not need to be large. On the other hand, if the number of paths is very large, then any adequate test set will need to be very large.

There is one issue we need to consider: since "all paths" is a strong criterion, it is possible that, even in the absence (or bounding) of loops, this criterion always results in an infeasibly large numbers of paths. In order to address this issue we also do an analysis of the number of paths in procedural programs (of equivalent size), and compare this with the number of paths for BDI programs (see Section 6).

Finally, it bears noting that the "all paths" criterion only considers which parts of the program were traversed during testing, but ignores the values of variables. So, for example, a trivial program consisting of the single statement $x := x * x$ has a single one-step path, which is trivially covered, but many traces ($x = 0, 1, 2 \ldots$).

1.2 HTN Planning

There are similarities between Hierarchical Task Network (HTN) planning (Erol, Hendler, & Nau, 1994) and BDI execution (de Silva & Padgham, 2004): both use a hierarchical representation with goals ("non-primitive tasks" in HTN terminology), plans ("decomposition methods") and goal-plan trees ("task networks"). The complexity of HTN planning has been explored. Given these similarities, can we simply exploit these known complexity results?

It turns out that we cannot do so, for the simple reason that the complexity of HTN planning concerns the plan *finding* problem, which is different to BDI plan *execution*, as Sardina and Padgham explain:

> BDI agent systems and HTN planners come from different communities and differ in many important ways. The former focus on the *execution* of plans, whereas the latter is concerned with the actual *generation* of such plans. The former are generally designed to *respond* to goals and information; the latter are designed to *bring about* goals. In addition, BDI systems are meant to be *embedded* in the real world and therefore take decisions based on a particular (current) state. Planners, on the other hand, perform *hypothetical reasoning* about actions and their interactions in multiple potential states. **Thus, failure has very different meaning for these two types of systems.** In the context of planning, failure means that a plan or potential plan is not suitable; within BDI agent systems failure typically means that an active (sub)plan ought to be

> aborted. Whereas backtracking upon failure is an option for planning systems, it is generally not for BDI systems, as actions are taken in the real world. (Sardina & Padgham, 2011, p. 45, bold emphasis added)

In other words, HTN systems plan ahead of execution, whereas BDI systems interleave execution and planning[2].

The HTN plan existence problem answers the question "does a plan exist?" and its complexity has been studied. In settings that correspond to BDI execution (many goals, total ordering within plans, and with variables) it is known to be EXPSPACE-hard and in DEXPTIME (Erol, Hendler, & Nau, 1994, 1996). However, this work does not address the question of BDI execution. When considering the complexity of plan existence in HTN planning we are asking about the computational complexity of a *search process* that will result in a plan. On the other hand, when we are asking about the number of paths in a goal-plan tree we are asking about the possibilities that arise when *executing* a plan.

To illustrate this point, consider the following example. Suppose we have a single goal G which can be decomposed into two alternative plans, P_1 and P_2. Plan P_1 consists of the sequential execution of actions a, b, and c; and plan P_2 consists of the sequential execution of actions d and e. The plan existence problem boils down to considering the options P_1 and P_2, since in this case the search space is very simple, offering only two options. On the other hand, the question of how many paths exist in BDI execution considers the different ways in which the goal-plan tree can be executed. Whereas HTN planning considers P_1 as a single atomic decomposition, BDI execution needs to consider the sequence of actions a, b, c as distinct steps. It is possible for all three actions to succeed (giving the trace a, b, c), but it also possible for action b to fail, followed by P_2 being (successfully) used (giving the trace $a, b\text{✘}, d, e$), or for action c to fail, followed by P_2 being (successfully) used (giving the trace $a, b, c\text{✘}, d, e$).

Overall, this means that the complexity analysis of Erol et al. (1994, 1996) is of a different problem, and that the HTN complexity results are not relevant. Finally, we note that, in fact, in our setting, the plan existence problem is actually trivially true: since BDI programs do not have constraints there is *always* an expansion of the program into a sequence of actions.

The remainder of this 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 number of paths in a BDI-style goal-plan tree. We then consider how our analysis and its assumptions hold up against a real system and a real platform (Section 5), and how our analysis of BDI programs compares with the same analysis (number of paths) of conventional procedural programs (Section 6). Finally, we conclude with a discussion of the implications for testing and future work (Section 7).

---

2. 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 (Paolucci, Shehory, Sycara, Kalp, & Pannu, 2000) has the ability to interleave planning and execution. However, no theoretical analysis of this extension has been reported, and the analysis of Erol, Hendler, and Nau (1994, 1996) applies to "classical" HTN planning.

2. The BDI Execution Model

Before we describe the Belief-Desire-Intention (BDI) model we explain why we chose this model of agent execution. 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.

The BDI model can be viewed from philosophical (Bratman, 1987) and logical (Rao & Georgeff, 1991) perspectives, but we are interested here in the *implementation* perspective, as exhibited in a range of architectures and platforms, such as JACK (Busetta, Rönnquist, Hodgson, & Lucas, 1999), JAM (Huber, 1999), dMARS (d'Inverno, Kinny, Luck, & Wooldridge, 1998), PRS (Georgeff & Lansky, 1986; Ingrand, Georgeff, & Rao, 1992), UM-PRS (Lee, Huber, Kenny, & Durfee, 1994), Jason (Bordini, Hübner, & Wooldridge, 2007), SPARK (Morley & Myers, 2004), Jadex (Pokahr, Braubach, & Lamersdorf, 2005) and IRMA (Bratman, Israel, & Pollack, 1988). 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 the work of Rao (1996), Winikoff, Padgham, Harland, and Thangarajah (2002) and Bordini et al. (2007), for example.

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 we assume[5] that a plan body is a sequence of steps, where each step is either an action[6] (which can succeed or fail) or an event to be posted.

For example, consider the simple plans shown in Figure 1. The first plan, Plan A, 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. This allows for additional plans for a given event to be added in a modular fashion, since the invoking

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 follows abstract notations such as AgentSpeak(L) (Rao, 1996) and Can (Winikoff et al., 2002) which aim to capture the essence of a range of (more complex) BDI languages.
6. 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.

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

Figure 1: Three Simple Plans

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)[7]:

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 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 as-

7. 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.

```
Boolean function execute(an-event)
let relevant-plans = set of plan instances resulting from
    matching all plans' event patterns to an-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
```

Figure 2: BDI Execution Cycle

sumptions 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 intention is clear from context.

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 (Bordini et al., 2007), 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 provides a "pattern" that allows the traditional BDI failure-handling mechanism to be specified succinctly (Bordini et al., 2007, pp. 171–172). Another alternative failure-handling approach is used by 2APL (Dastani, 2008) 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, 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 (Bratman et al., 1988).

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.

Note that "correct" (respectively "incorrect") behaviour is distinct from "successful" (respectively "failed") execution of a plan. Software testing is 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). On the other hand, BDI agents' behaviour traces are classified as being successful or failed. However, 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. This is a successful execution, but incorrect behaviour. 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 (finite) *goal-plan tree* into a sequence of action executions.

The events and plans can be visualised as a tree where each goal[8] 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[9] 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):

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

For example, the simple goal-plan tree shown in Figure 4 is modelled by the Prolog term `goal([plan([act(a)]), plan([act(b)])])`.

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

8. In order to be consistent with existing practice we shall use the term "goal" rather than "event" in the remainder of this paper.
9. In the remainder of this paper we will use the term "behaviour space size", rather than the more cumbersome term "number of paths through a BDI program".

```
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([X|Xs],X,Xs).
15 remove([X|Xs],Y,[X|Z]) :- remove(Xs,Y,Z).
```

Figure 3: Prolog Code to Expand Goal-Plan Trees

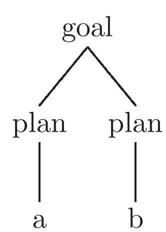


Figure 4: A Simple Goal-Plan Tree

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 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 because of a lack of resources, or other environmental issues.

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

4. Behaviour Space Size of BDI Agents

We now consider how many paths there are through a goal-plan tree that is being used by a BDI agent to realise a goal[10] using that 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 the behaviour space is 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. That is, all of the leaves of the goal-plan tree are the same distance (number of edges) away from the root of the tree. 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 $d-1$. Note that this definition of depth is the reverse of the usual definition (where the depth of the tree's root is defined as 0). We use this definition as it simplifies the presentation of the derivations later in this section.

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, or a goal may have a single relevant plan that has j applicable instances. Note that this assumption rules out the possibility of there being an infinite number of applicable plan instances, which would be the case if a plan's context condition has an infinite number of solutions. This cannot occur if the context condition is defined in terms of conjunctions over propositions that refer to a finite belief base. However, it can occur if the agent's context conditions can also make use of a Prolog-like knowledge base (as is the case in some agent-oriented programming languages, such as Jason or Goal). Nevertheless, since we deal with applicable plans, we don't model context conditions.

10. 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.

Figure 5 shows a uniform goal-plan tree of depth 2.

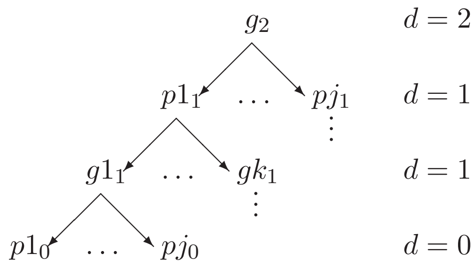


Figure 5: A uniform goal-plan tree

The assumptions made are clearly unrealistic. This means that we have to consider the possibility that real agent programs behave quite differently, since they do not meet these assumptions. We address this issue in a number of ways. Firstly, in Section 4.4 we consider a relaxation of the assumptions by defining semi-uniform trees, in which the number of available plan instances (j) can vary across different levels of the tree. Secondly, in Section 5.2 we consider an example of a (non-uniform) goal-plan tree from an industrial application. We derive the number of paths for this real goal-plan tree and compare it to the analysis of similarly-sized uniform goal-plan trees to see whether a real (non-uniform) tree has a significantly lower number of paths than a uniform tree. Finally, in Section 4.7, we consider the issue of infinite trees by allowing trees to be recursive, and defining the number of paths (up to a bound on the path length) of a recursive tree.

Our analysis uses the following terminology:

- Our uniformity assumptions mean that the structure of the subtree rooted at a goal or plan node is determined solely by its depth, and we can therefore denote a goal or plan node at depth d as g_d or p_d (respectively).
- We use $n^{\checkmark}(x_d)$ to denote the number of *successful* execution paths of a goal-plan tree of depth d rooted at x (where x is either a goal g or a plan p). Where specifying d is not important we will sometimes elide it, writing $n^{\checkmark}(x)$.
- Similarly, we use $n^{\times}(x_d)$ to denote the number of *unsuccessful* execution paths of a goal-plan tree of depth d with root x (either g or p).
- We extend this notation to *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 (for example, $g_1^3 = g_1; g_1; g_1$).

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 (2004, pp. 17–19).

Roughly speaking, the number of ways a goal 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. the goal is represented by an "or" node). On the other hand, the number of ways a plan 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. the plan is represented by an "and" node). 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[11]; then, because we select one of the children, 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 0) can be executed (successfully) in exactly one way. This yields the following definition:

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

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: (and if $k = 1$ we have $n^{\checkmark}(g_d) = n^{\checkmark}(p_d) = j^d$)

$$\begin{aligned} n^{\checkmark}(g_d) &= j^{(k^d-1)/(k-1)} && (1) \\ n^{\checkmark}(p_d) &= j^{k\,(k^d-1)/(k-1)} && (2) \end{aligned}$$

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, can be calculated as follows (Naish, 2007, Section 3):

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

11. 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 $j\,n$ rather than $n_1 + \ldots + n_j$.

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

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

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 those for 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, $n^{\text{✗}}(x_1; x_2) = n^{\text{✗}}(x_1) + n^{\checkmark}(x_1)\, n^{\text{✗}}(x_2)$; 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 $n^{\text{✗}}(x^k) = n^{\text{✗}}(x)\,(1 + \cdots + n^{\checkmark}(x)^{k-1})$, 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: $g_d; a^\ell$:

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

We then have for $d > 0$:

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

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.

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

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[12].

The number of executions can then be computed as follows: if a goal g_d has j applicable plan instances, each having $n^{\times}(p_{d-1})$ unsuccessful executions, then we have $n^{\times}(p_{d-1})^j$ unsuccessful executions of all of these plans in sequence. Since the plans can be selected in any order we multiply this by $j!$ yielding $n^{\times}(g_d) = j!\, n^{\times}(p_{d-1})^j$.

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:

$$\begin{aligned} n^{\times}(g_d) &= j!\, n^{\times}(p_{d-1})^j && (3)\\ n^{\times}(p_0) &= \ell && (4)\\ n^{\times}(p_d) &= \ell + (n^{\times}(g_d) + \ell\, n^{\checkmark}(g_d))\, \frac{n^{\checkmark}(g_d)^k - 1}{n^{\checkmark}(g_d) - 1} && (5)\\ & \quad (\text{for } d > 0 \text{ and } n^{\checkmark}(g_d) > 1) \end{aligned}$$

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: $pa, pbandpc$. 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, then pb is executed successfully
- pa fails, pb is then executed and fails, and then pc is executed and succeeds

This leads to a definition of the form

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

12. 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 as a result of there being no applicable plan instances. This is a conservative assumption: relaxing it results in the number of behaviours being even larger.

However, 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

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

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):

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

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

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

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 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 (Dastani, 2008) cannot be analysed in a general way, since it depends on the details of the specific agent program; and IRMA (Bratman et al., 1988) 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, plans and 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 5). 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 $G(n) = 1 + (j \times k) + (j \times k)^2 + \cdots + (j \times k)^{n-1}$. 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

Parameters			Number of				
j	k	d	goals	plans	actions	$n^{✔}(g)$	$n^{✘}(g)$
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 $n^{✔}(g)$ and $n^{✘}(g)$ **without failure handling**. The first number under "actions" (e.g. 62) is the number of actions in the tree, the second (e.g. 13) is the number of actions in a single execution where no failures occur.

Parameters			Number of				
j	k	d	goals	plans	actions	$n^{✔}(g)$	$n^{✘}(g)$
2	2	3	21	42	62 (13)	$\approx 6.33 \times 10^{12}$	$\approx 1.82 \times 10^{13}$
3	3	3	91	273	363 (25)	$\approx 1.02 \times 10^{107}$	$\approx 2.56 \times 10^{107}$
2	3	4	259	518	776 (79)	$\approx 1.82 \times 10^{157}$	$\approx 7.23 \times 10^{157}$
3	4	3	157	471	627 (41)	$\approx 3.13 \times 10^{184}$	$\approx 7.82 \times 10^{184}$

Table 2: Illustrative values for $n^{✔}(g)$ and $n^{✘}(g)$ **with failure handling**

$P_l(n)$ 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 $(1 \times 3 \times 10) + (1 \times 32) = 62$ actions.

4.4 Recurrence Relations

The equations in the previous sections define the functions $n^{✔}$ and $n^{✘}$ 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 j for any given goal. The aim of this section is to explore the effects of changing j. We do this by relaxing our uniformity assumption. Specifically, we allow 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 refer to these as *semi-uniform* goal-plan trees. We then derive a set of recurrence relations for $n^{✔}$ and $n^{✘}$ 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 $n^{✘}(g_{\mathbf{j}})$ and $n^{✔}(g_{\mathbf{j}})$ where $\mathbf{j}$ is a list[13] $(j_d, j_{d-1}, \ldots, j_0)$ in which each element j_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 $j \cdot \mathbf{j}$ to represent the list with head j and tail $\mathbf{j}$.

13. The order corresponds to our definition of depth, which decreases down the tree.

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 $f^{✗}$ and $f^{✓}$ of $n^{✗}(p_{\mathbf{j}})$ and (for $f^{✓}$) $n^{✓}(p_{\mathbf{j}})$. Our aim is to find a recursive definition of $f^{✗}$ and $f^{✓}$ as a recurrence on j.

$$\begin{aligned} n^{✗}(g_{j\cdot\mathbf{j}}) &= f^{✗}(j,\, n^{✗}(p_{\mathbf{j}})) \\ n^{✓}(g_{j\cdot\mathbf{j}}) &= f^{✓}(j,\, n^{✗}(p_{\mathbf{j}}),\, n^{✓}(p_{\mathbf{j}})) \end{aligned}$$

where

$$\begin{aligned} f^{✗}(j,a) &= j!\,a^j \\ f^{✓}(j,a,b) &= \sum_{i=1}^{j} b\,a^{i-1}\,\frac{j!}{(j-i)!} \\ &\quad \text{(change bounds on } i \text{ to } 0\ldots n\text{, hence replace } i \text{ by } i+1) \\ &= \sum_{i=0}^{j-1} b\,a^{(i+1)-1}\,\frac{j!}{(j-(i+1))!} \\ &\quad \text{(simplify using } (j-(i+1))! = (j-i)!/(j-i)\ ) \\ &= \sum_{i=0}^{j-1} b\,a^{i}\,\frac{j!(j-i)}{(j-i)!} \\ &\quad \text{(multiple by } i!/i! \text{ and reorder)} \\ &= \sum_{i=0}^{j-1} \frac{j!}{i!(j-i)!}\,i!\,a^i\,(j-i)\,b \\ &\quad \text{(use definition of binomial: } \binom{j}{i} = j!/i!(j-i)!) \\ &= \sum_{i=0}^{j-1} \binom{j}{i}\,i!\,a^i\,(j-i)\,b \end{aligned} \tag{9}$$

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

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

$$f^{✗}(j,a) = j!\,a^j = (j\,(j-1)\,\ldots\,2\cdot 1)\ \ (\underbrace{a\,a\,\ldots\,a\,a}_{j \text{ times}}) = (j\,a)\,((j-1)\,a)\,\ldots\,(2\,a)\,(1\,a)$$

$$
\begin{array}{rcl}
n^{\checkmark}(g_{j\cdot\mathbf{j}}) & = & f^{\checkmark}(j,\, n^{\times}(p_{\mathbf{j}}),\, n^{\checkmark}(p_{\mathbf{j}})) \\
n^{\times}(g_{j\cdot\mathbf{j}}) & = & f^{\times}(j,\, n^{\times}(p_{\mathbf{j}})) \\
\\
f^{\checkmark}(0,a,b) & = & 0 \\
f^{\checkmark}(j{+}1,a,b) & = & (j{+}1)\,(b + a\, f^{\checkmark}(j,a,b)) \\
f^{\times}(0,a) & = & 1 \\
f^{\times}(j{+}1,a) & = & (j{+}1)\,a\, f^{\times}(j,a) \\
\\
n^{\checkmark}(p_{\langle\rangle}) & = & 1 \\
n^{\times}(p_{\langle\rangle}) & = & \ell \\
n^{\checkmark}(p_{\mathbf{j}}) & = & n^{\checkmark}(g_{\mathbf{j}})^k, \;\; \text{for } \mathbf{j} \neq \langle\rangle \\
n^{\times}(p_{\mathbf{j}}) & = & \ell + \left(n^{\times}(g_{\mathbf{j}}) + \ell\, n^{\checkmark}(g_{\mathbf{j}})\right) \dfrac{n^{\checkmark}(g_{\mathbf{j}})^k - 1}{n^{\checkmark}(g_{\mathbf{j}}) - 1}, \;\; \text{for } \mathbf{j} \neq \langle\rangle
\end{array}
\tag{10}
$$

Figure 6: Recurrence relations for the numbers of failures and successes of a goal plan tree in the presence of failure handling

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

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.) (Wilf, 1994) for the sequence $\{f^{\checkmark}(j,a,b)\}_{j=0}^{\infty}$, and then using that to derive a recurrence relation. The details are given in Appendix B, and yield equation 10 in Figure 6.

Equation 10 (copied from Equation 25 in Appendix B) gives us the recurrence relation for the sequence $\{f^{\checkmark}(j,a,b)\}_{j=0}^{\infty}$ that we have been seeking[14]. Figure 6 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 gives us a different way of looking at the recurrence, and allows us to more easily see how the behaviour space grows as the number of applicable plans, j, 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 $f^{\checkmark}(j{+}1,a,b)$ can be seen to have the following combinatorial interpretation. One plan must be selected to try initially (there are $j{+}1$ 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 j plans, which can occur in $f^{\checkmark}(j,a,b)$ ways.

We can see that the growth in the number of successful executions for a goal grows at a rate greater than $j!a^j$, because of the presence of the b term. The relaxed uniformity

14. In the simple case when $a = b = 1$ this is listed as sequence A007526 in the On-Line Encyclopedia of Integer Sequences (Sloane, 2007): "the number of permutations of nonempty subsets of $\{1,\cdots,n\}$".

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 (with the exception of Section 4.7).

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 $p^{\checkmark}(x_d) = 1 - p^{\text{✗}}(x_d)$.

We assume that the probability of an action failing is ϵ_a[15]. Then the probability of a given plan's actions all succeeding is simply $(1 - \epsilon_a)^x$ where x is the number of actions. Hence the probability of a plan failing because of the failure of (one of) its actions is simply $1 - (1 - \epsilon_a)^x$, i.e. for a plan at depth 0 the probability of failure is:

$$\epsilon_0 = 1 - (1 - \epsilon_a)^{\ell}$$

and for a plan at depth greater than 0 the probability of failure due to actions is:

$$\epsilon = 1 - (1 - \epsilon_a)^{\ell\,(k+1)}$$

(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 $p^{\checkmark}(p_d) = (1 - \epsilon)\, p^{\checkmark}(g_d)^k$. 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 must select one plan and execute it, so the probability of success is the probability of that plan succeeding, i.e. $p^{\checkmark}(g_d) = p^{\checkmark}(p_{d-1})$. We ignore for the moment the possibility of a goal failing because there are no applicable plans. This assumption is relaxed later on.

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

$$\begin{aligned} p^{\text{✗}}(g_d) &= p^{\text{✗}}(p_{d-1}) \\ p^{\text{✗}}(p_0) &= \epsilon_0 \\ p^{\text{✗}}(p_d) &= 1 - \left[(1 - \epsilon)\,(1 - p^{\text{✗}}(g_d))^k\right] \end{aligned}$$

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

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

Table 3: Goal failure probabilities with and without failure handling

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

$$\begin{aligned} p^{\times}(g_d) &= p^{\times}(p_{d-1})^j \\ p^{\times}(p_0) &= \epsilon_0 \\ p^{\times}(p_d) &= 1 - \left[(1-\epsilon)\left(1 - p^{\times}(g_d)\right)^k\right] \end{aligned}$$

It is not easy to see from the equations what the patterns of probabilities actually are, and so, for illustration purposes, Table 3 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 $\epsilon_a = 0.05$ and hence $\epsilon \approx 0.185$ (which is rather high), and where $\epsilon_a = 0.01$ and hence $\epsilon \approx 0.04$.

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.

We now relax the assumption that a goal cannot fail because there are no applicable plans, i.e. that a goal will only fail once all plans have been tried. Unfortunately, relaxing this assumption complicates the analysis as we need to consider the possibility that none of the remaining plans are applicable at *each point* where failure handling is attemped.

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 because none of the remaining plans are applicable. For the case with no failure handling a non-zero ϵ_g indicates that there are situations where a goal does not have applicable plans, which may indicate an error on the part of the programmer, or that in certain situations a goal may not be possible to achieve. 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 $p^{\times}(g_d) = \epsilon_g + (1 - \epsilon_g)\, p^{\times}(p_{d-1})$, 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. We now have the following equations for the case without failure handling:

$$\begin{aligned} p^{\times}(g_d) &= \epsilon_g + (1-\epsilon_g)\, p^{\times}(p_{d-1}) \\ p^{\times}(p_0) &= \epsilon_0 \\ p^{\times}(p_d) &= 1 - [(1-\epsilon)\,(1-p^{\times}(g_d))^k] \end{aligned}$$

Observe that setting $\epsilon_g = 0$ 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 because no plans are applicable (ϵ_g)
- If there are applicable plans ($(1-\epsilon_g)\ldots$) 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 ($(1-\epsilon_g)\ldots$) and the selected plan fails ($p^{\times}(p_{d-1})$).

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

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

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 $(1-\epsilon_g)$, but the selected plan fails ($p^{\times}(p_{d-1})$) and then either there are no further applicable plans (ϵ_g), or

C. there are applicable plans $(1-\epsilon_g)$, but the selected plan fails ($p^{\times}(p_{d-1})$) 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:

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

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

$$\begin{aligned} p^{\times}(g_d) &= p^{\times}(g_d, j) \\ p^{\times}(g_d, 1) &= \epsilon_g + (1-\epsilon_g)\, p^{\times}(p_{d-1}) \\ p^{\times}(g_d, i+1) &= \epsilon_g + (1-\epsilon_g)\, p^{\times}(p_{d-1})\, p^{\times}(g_d, i) \\ p^{\times}(p_0) &= \epsilon_0 \\ p^{\times}(p_d) &= 1 - [(1-\epsilon)\,(1-p^{\times}(g_d))^k] \end{aligned}$$

		No failure handling			With failure handling		
ϵ_a	d	$\epsilon_g = 0$	$\epsilon_g = 0.01$	$\epsilon_g = 0.05$	$\epsilon_g = 0$	$\epsilon_g = 0.01$	$\epsilon_g = 0.05$
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	$\epsilon_g = 0$	$\epsilon_g = 0.005$	$\epsilon_g = 0.01$	$\epsilon_g = 0$	$\epsilon_g = 0.005$	$\epsilon_g = 0.01$
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%

Table 4: Goal failure probabilities with and without failure handling when goals can have no applicable plans

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

As before, it is not immediately clear from the formulae what the actual patterns of probability are. Considering illustrative examples, Table 4 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.

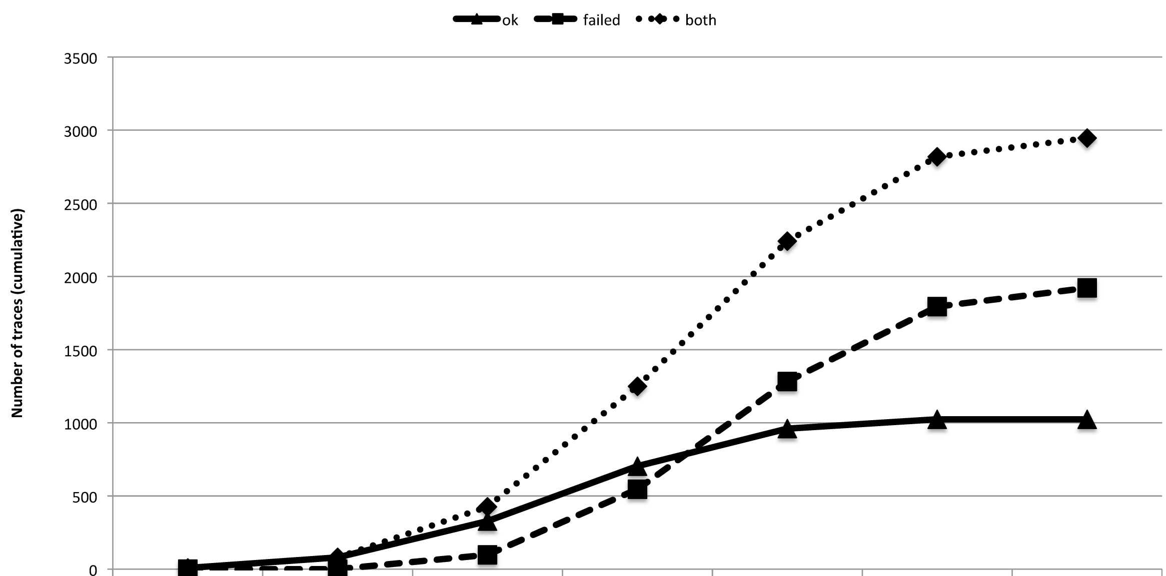
4.6 Analysis of the Rate of Failures

In this section we briefly examine how the number of traces for a goal-plan tree is affected by placing a bound on the rate of action failures that can occur within a trace. For simplicity, we work with uniform goal-plan trees, but the construction below extends trivially to semi-uniform goal-plan trees.

In Figure 6 we presented equations for calculating the total number of behaviours for a goal-plan tree (with failure handling). But how many of these behaviours involve a possibly unrealistic number of action failures? If we make an assumption that there is an upper limit to the rate of action failures[16], i.e. the number of failures divided by the length of the trace, how does this affect the number of possible 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 a high rate of action failure and how many involve only a small percentage of failures? Figure 7 contains (cumulative) counts that were generated by looking at all possible executions in this (small) case, plotted against the number of action failures. The x axis shows for a given value $N \in \{0, \ldots, 6\}$ how many traces there are 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. Figure 8 shows the equivalent graph for the *rate* of action failure: each trace has its failure rate computed (the number of failures divided by the length of the trace),

16. Bounding the rate of action failures allows us to model an assumption that the environment has limited unpredictability, or perhaps that the programmer has limited incompetence!



	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

Number of failures

Figure 7: Number of traces (cumulative) vs. number of failures for $j = k = 2$, $\ell = 1$, $d = 2$

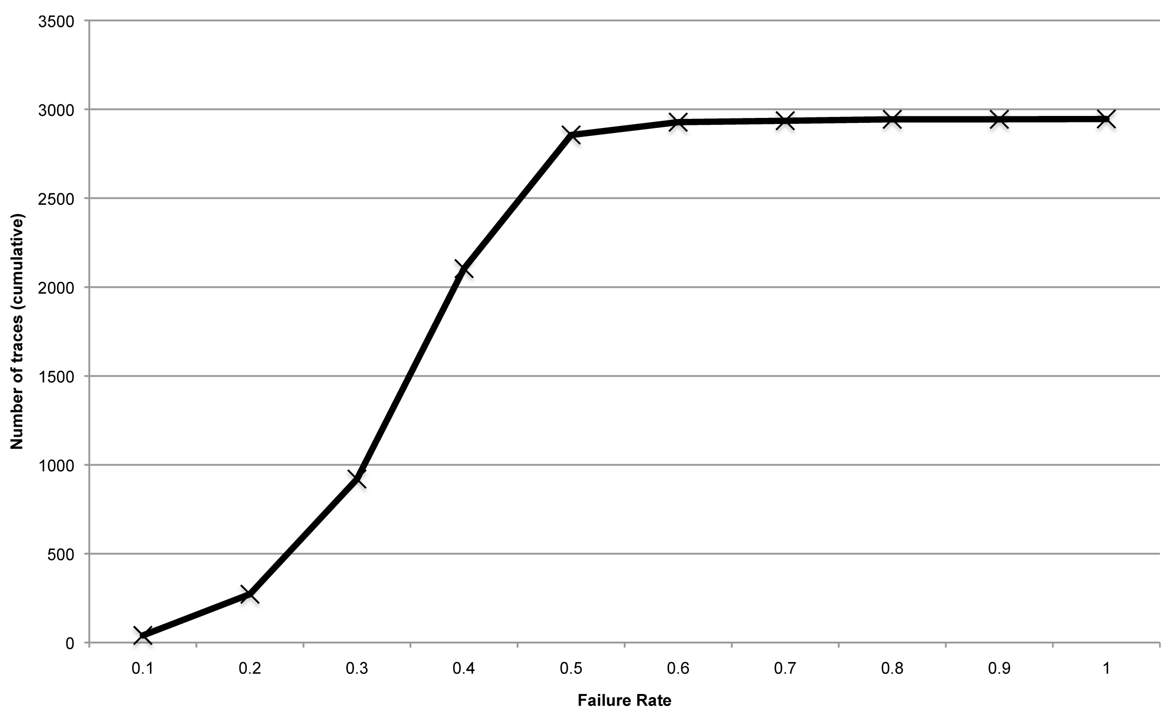


Figure 8: Number of traces (cumulative) vs. failure rate for $j = k = 2$, $\ell = 1$, $d = 2$

and the number of traces are counted for each range of failure rate. For instance, the first data point in the graph shows that there are 40 traces with a failure rate ≤ 0.1.

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.

For a given plan body segment[17] s (and particularly for $s = g_d$), we are interested in computing the numbers of successful and failed traces in which the failure rate is bounded by a given ratio r between the number of failed actions and the total number of actions, i.e. the proportion of actions in an execution trace that fail. We denote these by $n^{\checkmark}_{\leq r}(s)$ and $n^{\times}_{\leq r}(s)$. To compute these values, we first determine for integers $m > 0$ and $n \geq 0$ the numbers of successful and failed traces that have length m and contain exactly n action failures, denoted $n^{\checkmark}_r(s, m, n)$ and $n^{\times}_r(s, m, n)$, respectively. We define the length of a trace to be the number of actions (both successful and unsuccessful) that it contains. Note that for any finite goal-plan tree, such as a uniform or semi-uniform one, there is a maximum possible trace length and so $n^{\checkmark}_r(s, m, n)$ and $n^{\times}_r(s, m, n)$ can only be non-zero for a finite number of integer pairs (m, n) in the positive quadrant of the plane or on the positive m axis (in the case that $n = 0$). Once we have these values, we can calculate $n^{\checkmark}_{\leq r}(s)$ as the sum of all $n^{\checkmark}_r(s, m, n)$ for which $\frac{n}{m} \leq r$, and similarly for $n^{\times}_{\leq r}s$ using $n^{\times}_r(s, m, n)$.

We begin by considering *ordinary*[18] and *bivariate* generating functions (Wilf, 1994) for the values $n^{\checkmark}_r(s, m, n)$ and $n^{\times}_r(s, m, n)$:

$$\begin{aligned}
F^{\checkmark}_r(s, x, y) &= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} n^{\checkmark}_r(s, m, n)\, x^m y^n \\
F^{\times}_r(s, x, y) &= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} n^{\times}_r(s, m, n)\, x^m y^n
\end{aligned}$$

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

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

$$\begin{aligned}
F^{\checkmark}_r(s_1; s_2, x, y) &= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} n^{\checkmark}_r(s_1; s_2, m, n)\, x^m y^n \\
&= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \left(\sum_{p+q=m} \; \sum_{t+u=n} n^{\checkmark}_r(s_1, p, t)\, n^{\checkmark}_r(s_2, q, u) \right) x^m y^n
\end{aligned}$$

The double sum in parentheses considers, for any trace, all possible ways of allocating the number of actions m and number of action failures n to the (necessarily) successful executions of s_1 and s_2, and the sums are over non-negative integer values of p, q, t and u.

17. Recall that, as defined towards the start of Section 4 (page 83), a plan body segment s is a sequence $x_1; \ldots; x_n$ where each x_i is either a goal or an action.
18. Ordinary generating functions differ from exponential generating functions by not including denominators that are factorials of the powers of the variable(s).

We then have:

$$
\begin{aligned}
F_r^{\checkmark}(s_1;s_2,x,y) &= \sum_{m=0}^{\infty}\sum_{p+q=m}\sum_{n=0}^{\infty}\sum_{t+u=n} n_r^{\checkmark}(s_1,p,t)\,x^p y^t\, n_r^{\checkmark}(s_2,q,u)\,x^q y^u \\
&= \sum_{p=0}^{\infty}\sum_{q=0}^{\infty}\sum_{t=0}^{\infty}\sum_{u=0}^{\infty} n_r^{\checkmark}(s_1,p,t)\,x^p y^t\, n_r^{\checkmark}(s_2,q,u)\,x^q y^u \\
&= \left(\sum_{p=0}^{\infty}\sum_{t=0}^{\infty} n_r^{\checkmark}(s_1,p,t)\,x^p y^t\right)\left(\sum_{q=0}^{\infty}\sum_{u=0}^{\infty} n_r^{\checkmark}(s_2,q,u)\,x^q y^u\right) \\
&= F_r^{\checkmark}(s_1,x,y)\,F_r^{\checkmark}(s_2,x,y)
\end{aligned}
$$

The second line above is derived using the identity $\sum_{m=0}^{\infty}\sum_{p+q=m} f(p,q) = \sum_{p}^{\infty}\sum_{q}^{\infty} f(p,q)$. Both expressions sum over all non-negative integers p and q, but the first expression does this by first summing over all non-negative values m on the horizontal axis, and then summing over all pairs (p,q) of non-negative integers lying on a line with slope -1 that intersects the horizontal axis at m.

Considering $F_r^{\times}(s_1;s_2,x,y)$, we have:

$$
\begin{aligned}
F_r^{\times}(s_1;s_2,x,y) &= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} n_r^{\times}(s_1;s_2,m,n)\,x^m y^n \\
&= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\Bigg(n_r^{\times}(s_1,m,n) \\
&\qquad + \sum_{p+q=m}\sum_{t+u=n} n_r^{\checkmark}(s_1,p,t)\,n_r^{\times}(s_2,q,u)\Bigg)\,x^m y^n \\
&= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} n_r^{\times}(s_1,m,n)\,x^m y^n \\
&\qquad + \sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\left(\sum_{p+q=m}\sum_{t+u=n} n_r^{\checkmark}(s_1,p,t)\,n_r^{\times}(s_2,q,u)\right)x^m y^n \\
&= F_r^{\times}(s_1,x,y) \\
&\qquad + \left(\sum_{p=0}^{\infty}\sum_{t=0}^{\infty} n_r^{\checkmark}(s_1,p,t)\,x^p y^t\right)\left(\sum_{q=0}^{\infty}\sum_{u=0}^{\infty} n_r^{\times}(s_2,q,u)\,x^q y^u\right) \\
&= F_r^{\times}(s_1,x,y) + F_r^{\checkmark}(s_1,x,y)\,F_r^{\times}(s_2,x,y)
\end{aligned}
$$

The second line above is based on the observation that each failed execution of $s_1;s_2$ of length m and with n action failures is either a failed execution of s_1 of length m and with n action failures occurring in that execution, or is a successful execution of s_1 of length p and with t failures followed by a failed execution of s_2 of length q and with u failures, where $p+q=m$ and $t+u=n$.

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

$$F_r^{\checkmark}(p_d, x, y) = F_r^{\checkmark}(a^\ell; (g_d; a^\ell)^k, x, y)$$
$$F_r^{\times}(p_d, x, y) = F_r^{\times}(a^\ell; (g_d; a^\ell)^k, x, y)$$

It remains to define $F_r^{\checkmark}(g_d, x, y)$ and $F_r^{\times}(g_d, x, y)$ in terms of $F_r^{\checkmark}(p_{d-1}, x, y)$ and $F_r^{\times}(p_{d-1}, x, y)$. To count the successful executions of g_d of length m and 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 m actions in the trace and 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 $F_r^{\checkmark}(g_d, x, y)$:

$$\begin{aligned}
&F_r^{\checkmark}(g_d, x, y)\\
&= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} n_r^{\checkmark}(g_d, m, n)\, x^m y^n\\
&= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\left(j\sum_{p=0}^{j-1}\binom{j-1}{p}p! \sum_{\ell_0+\cdots+\ell_p=m}\ \sum_{f_0+\cdots+f_p=n} n_r^{\checkmark}(p_{d-1}, \ell_0, f_0)\prod_{i=1}^{p} n_r^{\times}(p_{d-1}, \ell_i, f_i)\right) x^m y^n\\
&= j\sum_{p=0}^{j-1}\binom{j-1}{p}\, p! \sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\left(\sum_{\ell_0+\cdots+\ell_p=m}\ \sum_{f_0+\cdots+f_p=n} n_r^{\checkmark}(p_{d-1}, \ell_0, f_0)\prod_{i=1}^{p} n_r^{\times}(p_{d-1}, \ell_i, f_i)\right) x^m y^n\\
&= j\sum_{p=0}^{j-1}\binom{j-1}{p}\, p! \left(\sum_{\ell=0}^{\infty}\sum_{f=0}^{\infty} n_r^{\checkmark}(p_{d-1}, \ell, f)\, x^\ell y^f\right)\left(\sum_{\ell=0}^{\infty}\sum_{f=0}^{\infty} n_r^{\times}(p_{d-1}, \ell, f)\, x^\ell y^f\right)^p\\
&= j\sum_{p=0}^{j-1}\binom{j-1}{p}\, p!\, F_r^{\checkmark}(p_{d-1}, x, y)\, F_r^{\times}(p_{d-1}, x, y)^p
\end{aligned}$$

Constructing $F_r^{\times}(g_d, x, y)$ 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 $F_r^{\times}(g_d, x, y)$:

$$\begin{aligned}
F_r^{\times}(g_d, x, y) &= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} n_r^{\times}(g_d, m, n)\, x^m y^n\\
&= \sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\left(j! \sum_{\ell_1+\cdots+\ell_j=m}\ \sum_{f_1+\cdots+f_j=n} n_r^{\times}(p_{d-1}, \ell_1, f_1)\cdots n_r^{\times}(p_{d-1}, \ell_j, f_j)\right) x^m y^n\\
&= j!\left(\sum_{\ell=0}^{\infty}\sum_{f=0}^{\infty} n_r^{\times}(p_{d-1}, \ell, f)\, x^\ell y^f\right)^j\\
&= j!\, F_r^{\times}(p_{d-1}, x, y)^j
\end{aligned}$$

The equations above define a recursive procedure for computing $F_r^{\checkmark}(g_d, x, y)$ and $F_r^{\times}(g_d, x, y)$ for given values of d, j, k and ℓ. As discussed earlier in this section, given a way of calculating $n_r^{\checkmark}(s, m, n)$, we can calculate $n_{\leq r}^{\checkmark}(s)$ as a sum of all $n_r^{\checkmark}(s, m, n)$ for which $\frac{n}{m} \leq r$, and similarly for $n_{\leq r}^{\times} s$. We have used the Python rmpoly and GMPY2 libraries to generate polynomial representations of the functions $F_r^{\checkmark}(g_d, x, y)$ and $F_r^{\times}(g_d, x, y)$ for any specified values of d, j, k and l, and to calculate $n_{\leq r}^{\checkmark}(s)$ and $n_{\leq r}^{\times}(s)$ for various ratios r[19]. Figure 9 shows the results for $d = j = k = 3$ and $\ell = 1$.

Examining Figure 9 we can conclude two things. On the one hand, the number of traces really explodes for larger rates of action failures. For example, in Figure 9 most traces have a failure rate greater than 0.4. On the other hand, although making assumptions about the failure rate does reduce the number of possible traces, the number of traces is still quite large (note the scale on the y-axis). For instance, for a failure rate of ≤ 0.1 there are around 4.8×10^{44} failed executions and 8.7×10^{47} successful executions. For a failure rate of ≤ 0.2 the respective numbers are 1.0×10^{77} and 6.7×10^{77}, and for a failure rate of ≤ 0.3 they are 1.2×10^{96} and 2.7×10^{96}.

The shape of Figure 9 can be explained as follows. Firstly, the occurrence of an action failure triggers further activity (alternative plans), so more failures result in longer traces. Secondly, there are more longer traces than there are shorter traces, simply because the longer the trace, the more possibilities there are for variations (e.g. different orders of trying plans). This explains why the increase in Figure 9 starts off slowly and then accelerates: as we get more failures, we have longer traces, and for these longer traces there are more of them. In other word, if we were to plot the non-cumulative number of paths against the ratio of action failures we would see an initial increase: as the ratio grows, there are more paths. What this doesn't explain is why beyond a certain point we get fewer traces, and the cumulative graph levels out. The explanation here is quite simple: beyond a certain ratio (which appears to be around 0.4) there are no successful traces, and the number of failed traces also declines.

4.7 Recursive Trees

In Section 4.4 we developed recurrence relations that allowed us to relax the assumption that goal-plan trees are uniform, and considered semi-uniform trees. In this section we relax the assumption that goal-plan trees are finite, and we also allow trees to have any shape. We do this by considering arbitrary trees that are allowed to contain labels that refer to other parts of the tree, i.e. we allow trees to be recursive. We then derive generating functions, which can be seen as an extension of those derived in the previous section, for the number of paths (both successful and unsuccessful) in executing these recursive goal-plan trees. Obviously, an infinite tree has an infinite number of paths, and so we define generating functions that take as a parameter a *bound* on the lengths of the paths counted.

19. There are a finite number of actions that can be attempted during any execution of a goal-plan tree, and this bounds the length of its possible traces and the number of action failures that can occur within them. Thus $F_r^{\checkmark}(g_d, x, y)$ and $F_r^{\times}(g_d, x, y)$ are polynomials of finite order—only a finite number of coefficients are non-zero in the infinite sums that define them.

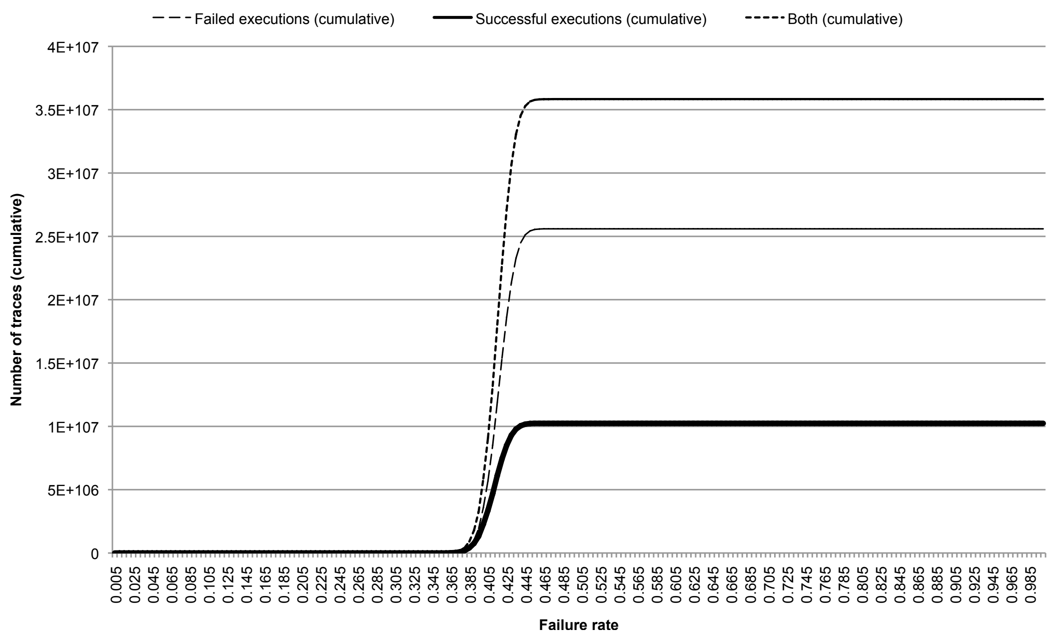


Figure 9: Number of traces (cumulative) vs. failure rate for $j = k = d = 3$ and $\ell = 1$

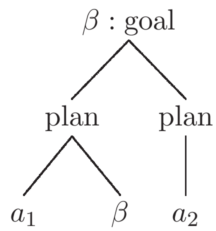
For a given upper bound on path length λ the equations specify the number of paths that have at most that many actions[20].

We begin by defining some notation for representing recursive trees: goals, plan-body multisets, plans, variables and bindings. A *goal* is represented by a term of the form *goal*(*plan-body-multiset*) where *plan-body-multiset* is a multiset representing the different applicable plan instances that can be used to satisfy the goal. This is a multiset because for our combinatorial analysis, only the structure of plans is significant. Therefore we use a single abstract action a to represent all actions[21], and a goal may be achievable using multiple plan instances that have the same structure, but which we must treat as distinct. We only need to represent the bodies of the plan instances, so each element in the multiset (i.e. *plan*) is a sequence of terms separated by the right-associative sequential composition operator ";". Each term in the sequence is either the abstract action term a, a goal term as defined above (representing a sub-goal), or a label (see below). Formally, a *plan-body multiset* PM is a multiset of plans, written $\{p_1{:}c_1, \ldots, p_j{:}c_j\}$ where each of the c_i is the number of times that the associated plan p_i appears in the multiset. We define the following multiset operations: $set(PM)$ is a set of all the p_i in the multiset PM, $\chi_{PM}(p_i)$ is the characteristic function denoting the number of times plan p_i appears in the multiset (i.e. c_i); and $PM_1 - PM_2$ is multiset subtraction, defined as $\chi_{PM_1 - PM_2}(x) = \max(\chi_{PM_1}(x) - \chi_{PM_2}(x), 0)$. Finally $|PM|$ is the size of the multiset, i.e. the sum of the c_i.

20. We can also use the equations derived in this section for non-recursive trees, in which case we allow $\lambda = \infty$, where we define $\infty - 1 = \infty$ and $F \restriction_{power(x) \leq \infty} = F$.
21. However, to avoid confusion, we will use numeric subscripts $(a_1, a_2, \ldots)$ to distinguish different occurrences of actions.

In order to allow recursive trees to be represented, it is possible for a step in a plan to be a *label* (denoted β, β_i or β') referring to a term in the provided *binding*, which is simply a mapping from labels to terms (either goal or plan terms). If b is a binding then we write $b[\beta]$ to denote the item that β is mapped to in b, or $\perp$ if there is no entry for β in b.

For example, consider the simple tree below, consisting of a goal with two plans, together with a binding that maps the variable β to the root of the tree. The first plan (on the left) has two steps: an action (a_1), and a recursive reference to the root of the tree (β). The second plan (on the right) just has a single action (a_2).



This recursive tree can be represented as follows. We define the binding $b = \{\beta \mapsto goal(\{(a_1;\beta){:}1, a_2{:}1\})\}$ which maps β to the whole tree, and then the tree itself is just β.

Before we proceed to defining generating functions, we introduce some auxiliary notation. If P is a power series then we use the standard notation $[x_1^{p_1} \cdots x_n^{p_n}]P$ to denote the coefficient of the term $x_1^{p_1} \cdots x_n^{p_n}$ in the series. We define $P \restriction_{cond}$ to denote the power series containing all the terms in P that satisfy the condition *cond*. We define $f \overset{x\leq n}{\times} g$ as $(f \times g) \restriction_{\text{power}(x)\leq n}$, i.e. $f \times g$ with any terms having a power of x greater than n removed. We define $f^{m \restriction x^{\leq n}}$ as $(f)^m \restriction_{\text{power}(x)\leq n}$, i.e. $(f)^m$ with any terms having a power of x greater than n removed.[22]

We are now in a position to derive generating functions that specify the number of paths through an arbitrary, and possible recursive, goal-plan tree, given a bound λ on the path length. We define s to be the BDI program represented as a term (i.e. goal, plan, plan multiset, action, or label), and b to be a binding mapping labels to terms (as defined above). We then define $n^{\checkmark}_{\infty}(s, m, n, b)$ to be the number of successful paths, with respect to s and binding b, that have m actions, n of which are failed actions. Similarly we define $n^{\times}_{\infty}(s, m, n, b)$ to be the number of *failed* paths, with respect to s and b, that have m actions, n of which are failed actions. We now want to derive recurrence relations for the generating functions[23]:

$$\begin{aligned} F^{\checkmark}_{\infty}(s, x, y, b, \lambda) &= \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n^{\checkmark}_{\infty}(s, m, n, b) x^m y^n \\ F^{\times}_{\infty}(s, x, y, b, \lambda) &= \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n^{\times}_{\infty}(s, m, n, b) x^m y^n \end{aligned}$$

where λ is an upper bound on the number of actions in a path.

22. This, and the previous operation, are directly supported by the rmpoly Python library for multivariate polynomials and series, which we have used to compute these generating functions.

23. The subscript ∞ is used to distinguish these generating functions, that allow for a recursive tree, from other generating functions defined elsewhere in the paper.

In order to simplify the presentation, the details of the more complex derivations are given in Appendix C. The resulting equations are shown in Figure 10. The first two equations (Equations 11 and 12 in Figure 10), which are applicable for any term t, capture the assumption that $\lambda > 0$ (and the remaining equations only apply when $\lambda > 0$). The next two equations simply specify that labels are looked up in the provided binding. Equation 15 indicates that there is a single successful path through action a, and that it has a single action and no unsuccessful actions (i.e. the generating function is $1x^1y^0$). Equation 16 similarly indicates that there is a single unsuccessful path through a single action a, which, unsurprisingly, has a single unsuccessful action (so the generating function is $1x^1y^1$).

Equations 17 and 18 deal with sequences: for a sequence $s_1; s_2$ to succeed both s_1 and s_2 must succeed, and we count the paths by concatenating sub-paths, which corresponds to multiplying power series. A sequence $s_1; s_2$ can fail if either s_1 fails, or if s_1 succeeds and s_2 then fails (alternatives correspond to the addition of power series). For both equations there is a special case: if s_1 is an action, then we can divide the overall path-length limit λ precisely: s_1 must have a trace of length 1 (since it is an action) and s_2 must therefore have a maximum length of $\lambda - 1$.

Having dealt with labels (β), single actions, and sequences, we next turn to goals (equations 19 and 20). In both cases the derivation is complex, and is covered in Appendix C. For $F^{\checkmark}_{\infty}$ (Equation 19 and Appendix C.1), the intuition is that a successful path through a goal's execution involves a single successful plan p, and some number of failed executions of plan selected from the remaining multiset of plans ($PM - \{p{:}1\}$). In the case where a plan appears more than once in the multiset, then we can select any of its occurrences, hence the multiplication by $\chi_{PM}(p)$. For the number of failed paths of a goal (Equation 20 and Appendix C.2) we introduce an auxiliary generating function $G^{\times}_{\infty}(PM, x, y, z, b, \lambda)$, where PM is a multiset of plans, and z is a variable whose power z^o indicates the *exact* number of plans from PM that are used. In other words, given the power series denoted by $G^{\times}_{\infty}(PM, x, y, z, b, \lambda)$, the term $c_{mno}x^m y^n z^o/o!$ indicates that there are c_{mno} paths that involve m actions, n of which failed, and exactly o of the plans in PM. The generating function $G^{\times}_{\infty}$ is a technical device that allows us to derive the definition of $F^{\times}_{\infty}$ that we need. Given this power series, the definition of $F^{\times}_{\infty}$ simply selects the terms that have $|PM|$ as the power of z (since all plans must fail for a goal to fail) using $\upharpoonright$ and then removes the $z^{|PM|}$ terms by dividing. Because $G^{\times}_{\infty}$ is an exponential generating function in z, which means that it includes a division by a factorial, we need to multiply by the factorial $|PM|!$ to remove it.

Equation 21 defines $F^{\times}_{\infty}(PM, x, y, b, \lambda)$, which is used in Equation 19, in terms of the auxiliary function $G^{\times}_{\infty}$. Its derivation is given in Appendix C.3. The intuition here is that for each possible number of plans that could be used (o) we limit the power series $G^{\times}_{\infty}$ to that value of o, and remove the z^o by dividing. The $o!$ is due to $G^{\times}_{\infty}$ being an exponential generating function in z (see Appendix C).

Finally, Equations 22 and 23 give the definition of $G^{\times}_{\infty}(PM, x, y, z, b, \lambda)$ (see Appendix C.4 for the derivation). Intuitively, Equation 22 creates the power series for each plan type, and then combines them (using $\overset{x \leq \lambda}{\times}$). Equation 23 is a little more complex: there is a single way of failing (when no plans are used, corresponding to the term $x^0y^0z^0 = 1$). Otherwise we can select any o of the c plans, and each of the plans must fail (corresponding to the term

$$
\begin{aligned}
F^{\checkmark}_{\infty}(t,x,y,b,\lambda) &= 0 \text{ if } \lambda \leq 0 && (11)\\
F^{\times}_{\infty}(t,x,y,b,\lambda) &= 0 \text{ if } \lambda \leq 0 && (12)\\
F^{\checkmark}_{\infty}(\beta,x,y,b,\lambda) &= F^{\checkmark}_{\infty}(b[\beta],x,y,b,\lambda) && (13)\\
F^{\times}_{\infty}(\beta,x,y,b,\lambda) &= F^{\times}_{\infty}(b[\beta],x,y,b,\lambda) && (14)\\
F^{\checkmark}_{\infty}(a,x,y,b,\lambda) &= x && (15)\\
F^{\times}_{\infty}(a,x,y,b,\lambda) &= xy && (16)
\end{aligned}
$$

$$
F^{\checkmark}_{\infty}(s_1;s_2,x,y,b,\lambda) = \begin{cases} F^{\checkmark}_{\infty}(s_1,x,y,b,1)\, F^{\checkmark}_{\infty}(s_2,x,y,b,\lambda-1) & \text{if } s_1 \text{ is an action} \\ F^{\checkmark}_{\infty}(s_1,x,y,b,\lambda) \overset{x\leq\lambda}{\times} F^{\checkmark}_{\infty}(s_2,x,y,b,\lambda) & \text{otherwise} \end{cases} \quad (17)
$$

$$
F^{\times}_{\infty}(s_1;s_2,x,y,b,\lambda) = \begin{cases} F^{\times}_{\infty}(s_1,x,y,b,1) + F^{\checkmark}_{\infty}(s_1,x,y,b,1)\, F^{\times}_{\infty}(s_2,x,y,b,\lambda-1) & \text{if } s_1 \text{ is an action} \\ F^{\times}_{\infty}(s_1,x,y,b,\lambda) + F^{\checkmark}_{\infty}(s_1,x,y,b,\lambda) \overset{x\leq\lambda}{\times} F^{\times}_{\infty}(s_2,x,y,b,\lambda) & \text{otherwise} \end{cases} \quad (18)
$$

$$
F^{\checkmark}_{\infty}(goal(PM),x,y,b,\lambda) = \sum_{p \in set(PM)} \chi_{PM}(p) F^{\checkmark}_{\infty}(p,x,y,b,\lambda) \overset{x\leq\lambda}{\times} F^{\times}_{\infty}(PM - \{p{:}1\},x,y,b,\lambda) \quad (19)
$$

$$
F^{\times}_{\infty}(goal(PM),x,y,b,\lambda) = |PM|! \frac{G^{\times}_{\infty}(PM,x,y,z,b,\lambda) \restriction_{power(z)=|PM|}}{z^{|PM|}} \quad (20)
$$

$$
F^{\times}_{\infty}(PM,x,y,b,\lambda) = \sum_{o=0}^{|PM|} o! \frac{G^{\times}_{\infty}(PM,x,y,z,b,\lambda) \restriction_{\text{power}(z)=o}}{z^{o}} \quad (21)
$$

$$
G^{\times}_{\infty}(\{p_1{:}c_1,\ldots,p_j{:}c_j\},x,y,z,b,\lambda) = G^{\times}_{\infty}(\{p_1{:}c_1\},x,y,z,b,\lambda) \overset{x\leq\lambda}{\times} \cdots \overset{x\leq\lambda}{\times} G^{\times}_{\infty}(\{p_j{:}c_j\},x,y,z,b,\lambda) \quad (22)
$$

$$
G^{\times}_{\infty}(\{p{:}c\},x,y,z,b,\lambda) = 1 + \sum_{o=1}^{c} \binom{c}{o} F^{\times}_{\infty}(p,x,y,b,\lambda)^{o \restriction x\leq\lambda}\, z^{o} \quad (23)
$$

Figure 10: Equations for Recursive Goal-Plan Trees

$F^{\times}_{\infty}(p,x,y,b,\lambda)$), giving the number of failed traces across these o plans as:

$$
F^{\times}_{\infty}(p,x,y,b,\lambda)^{o \restriction x\leq\lambda} = \underbrace{F^{\times}_{\infty}(p,x,y,b,\lambda) \overset{x\leq\lambda}{\times} \cdots \overset{x\leq\lambda}{\times} F^{\times}_{\infty}(p,x,y,b,\lambda)}_{o \text{ times}}
$$

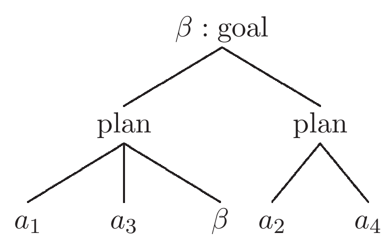
We have used the Python rmpoly and GMPY2 libraries to generate polynomial representations of the functions $F^{\checkmark}_{\infty}(t,x,y,b,\lambda)$ and $F^{\times}_{\infty}(t,x,y,b,\lambda)$ (as defined in Figure 10) for any specified values of t, b, and λ. Then we defined a simple tree (the one given earlier in this section as an example) and computed the number of paths for different values of λ.

The values of λ have been chosen to correspond to values in Table 2 (which is where the values for $n^{\checkmark}(g)$ and $n^{\times}(g)$ come from[24]). In Table 2, the values of 62 and 363 correspond to the longest path, and so we argue that when comparing a recursive tree to a uniform tree, we should consider the same path length limit. The results are shown in Table 5.

λ	$n^{\checkmark}(g)$	$n^{\times}(g)$	$n^{\checkmark}_{\infty}(s)$	$n^{\times}_{\infty}(s)$
62	$\approx 6.33 \times 10^{12}$	$\approx 1.82 \times 10^{13}$	$\approx 3.8 \times 10^{13}$	$\approx 4.3 \times 10^{9}$
363	$\approx 1.02 \times 10^{107}$	$\approx 2.56 \times 10^{107}$	$\approx 1.9 \times 10^{76}$	$\approx 6.1 \times 10^{54}$

Table 5: Comparing $n^{\checkmark}$ and $n^{\checkmark}_{\infty}$ (respectively $n^{\times}$ and $n^{\times}_{\infty}$).

Looking at the numbers in Table 5, it is worth noting that the recursive tree that we have used is extremely simple: two plans, each with only a single action. The low number of actions (and sparseness of the tree) account for the relatively low number of unsuccessful paths. For instance, if we modify the tree by adding extra actions (giving the tree and binding below) then for $\lambda = 62$ there are around 3.9×10^{13} successful paths, and 1.5×10^{11} unsuccessful paths. Unfortunately, Python was unable to calculate $n^{\checkmark}_{\infty}$ or $n^{\times}_{\infty}$ for this tree with $\lambda = 363$, but it did manage $\lambda = 362$, for which there are 1.26×10^{64} successful paths, and 3.28×10^{63} unsuccessful paths. This shows, as expected, that the number of unsuccessful paths is higher for the more complex tree. That there are fewer successful paths for the more complex tree can be explained by observing that, for this tree, traces are longer (more actions need to be done), and so more of the traces are excluded by the bound on trace length λ.



Overall, the analysis in this section, and its application to $\lambda = 62$ and 363 confirms that the number of paths in a recursive tree depends on the tree's structure (which is unsurprising), but also indicates that even for a very simple recursive tree, the number of paths for a given upper bound on path length quickly becomes extremely large.

5. A Reality Check

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

In this section we consider the two issues of whether this analysis is faithful, and whether it is applicable to real systems. The analysis made a number of simplifying assumptions,

24. They correspond to the first two rows of the table, which respectively involve 62 and 363 actions.

and these mean that the results may not be faithful to the semantics of a real BDI platform, or that they may not apply to real systems. We thus conduct two "reality checks" to assess whether our analysis is faithful (Section 5.1) and whether it is applicable (Section 5.2).

We firstly assess whether our analysis is faithful to real BDI platforms, i.e. that it does not omit significant features, or contain errors. We do this by comparing our abstract BDI execution model with results from a real BDI platform, namely JACK (Busetta et al., 1999). 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. This comparison is, in essence, a basic reality check: we are simply checking that the analysis in the previous section does indeed match the execution semantics of a typical BDI platform. We do this by modelling an artificial goal-plan tree in the BDI platform.

Next, in order to assess to what extent our analysis results apply to real systems, 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 Appendix A in the JACK agent programming language[25]. The structure of the plans and events[26] 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):

```
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

25. The code is available upon request from the authors.
26. JACK models a goal as a "`BDIGoalEvent`".

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	$\approx 2.98 \times 10^{20}$	$\approx 9.69 \times 10^{20}$
(*) The paper says 60 goals,					294,912	1,625,302	$\approx 6.28 \times 10^{15}$	$\approx 8.96 \times 10^{15}$
but Figure 11 has 57 goals.					294,912	812,651	$\approx 9.66 \times 10^{11}$	$\approx 6.27 \times 10^{11}$

Table 6: Illustrative values for $n^{\checkmark}(g)$ and $n^{\times}(g)$ (bottom part is $\ell = 4$ in first row, $\ell = 2$ in second, and $\ell = 1$ in last row)

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 an industrial application at Daimler which used BDI agents to realise agile business processes (Burmeister, Arnold, Copaciu, & Rimassa, 2008). 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 11 shows[27] a goal-plan tree from the work of Burmeister et al. (2008) which has "*60 achieve goals in up to 7 levels. 10 maintain goals, 85 plans and about 100 context variables*" (Burmeister et al., 2008, p. 41). Unlike the typical goal-plan trees used in BDI platforms, the tree in Figure 11 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 6 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 Burmeister et al. (2008) that their plans are in fact

27. The details are not meant to be legible: the structure is what matters.

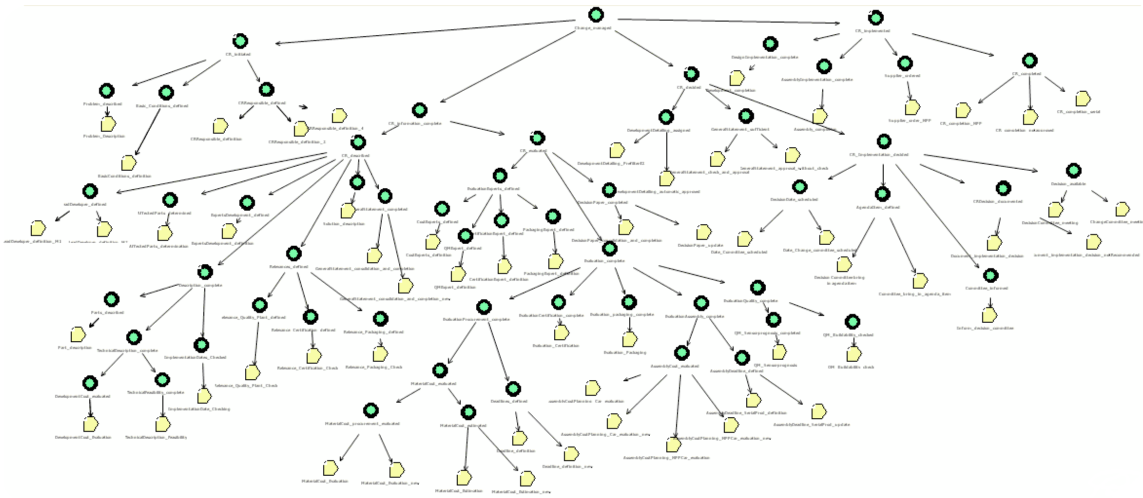


Figure 11: Goal-plan tree from the work of Burmeister et al. (2008, Figure 6) (reproduced with permission from IFAAMAS)

more complicated, and can contain nested decision making (e.g., see Burmeister et al., 2008, 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 11 has size in between the first two rows of Table 6. Comparing the number of possible behaviours of the uniform goal-plan trees against the real (and non-uniform) goal-plan tree, we see that the behaviour space is somewhat smaller in the real tree, but that it is still quite large, especially in the case with failure handling. However, we do note the following points:

1. The tree of Figure 11 only has plans at the leaves, which reduces its complexity. In other words a goal-plan tree that was more typical in having plans alternating with goals would have a larger number of possible behaviours.

2. The figures for the tree are a conservative estimate, since we assume that leaf plans have only simple behaviour. In other words, the number of paths calculated is an under-estimate of the actual number of paths in the real application.

6. Comparison with Procedural Programs

In order to argue that BDI programs are *harder* to test than non-agent programs, we need a comparison. Specifically, we need to analyze the number of paths in non-agent programs, and compare with those in agent programs. This will allow us to address the concern that the "all paths" criterion for test suite adequacy always requires an infeasibly large number of tests. This section briefly does this, by analyzing the number of paths in a procedural program.

Number of actions / statements	$n(m)$	$n^{✔}(g)$	$n^{✘}(g)$
62	6,973,568,802	$\approx 6.33 \times 10^{12}$	$\approx 1.82 \times 10^{13}$
363	$\approx 5.39 \times 10^{57}$	$\approx 1.02 \times 10^{107}$	$\approx 2.56 \times 10^{107}$
776	$\approx 2.5 \times 10^{123}$	$\approx 1.82 \times 10^{157}$	$\approx 7.23 \times 10^{157}$
627	$\approx 5.23 \times 10^{99}$	$\approx 3.13 \times 10^{184}$	$\approx 7.82 \times 10^{184}$

Table 7: Comparison of values between $n(m)$, $n^{✔}(g)$ and $n^{✘}(g)$.

We define a program as being composed of primitive statements s, sequences of statements $P_1; P_2$, or conditionals that select between two sub-programs. Since we do not capture the conditions of statements, we elide the condition, and write a conditional as $P_1 + P_2$ indicating that one of the P_i is selected. Note that, as for BDI analysis, we exclude loops.

We define the number of paths in a program P as $n(P)$. It is straightforward[28] to see that the definition of $n(P)$ is:

$$\begin{aligned} n(s) &= 1 \\ n(P_1; P_2) &= n(P_1) \times n(P_2) \\ n(P_1 + P_2) &= n(P_1) + n(P_2) \end{aligned}$$

In order to compare with BDI programs, we consider the size of the program, and compare programs of the same size. The key question then is: does a procedural program with m nodes have significantly fewer paths than a BDI program of the same size? We define the size of a program P as the number of primitive statements it contains, and denote it $|P|$. Note that this means that we do not count the "internal" nodes of the syntax tree (i.e. the "+" or ";"). Therefore, when comparing with BDI programs, we consider the size of a BDI program to be the number of actions[29].

We now work out how the number of paths varies with the size of the program P. If m is the size of a program (and therefore a natural number), then we define $n(m) \equiv \max\{n(P) : |P| = m\}$. That is, $n(m)$ is the largest number of paths possible for a program of size m.

Appendix D contains the derivation of $n(m)$, resulting in the following definition (where $m \geq 6$ is a multiple of 3):

$$\begin{array}{lcllcl} n(1) & = & 1 & n(2) & = & 2 \\ n(3) & = & 3 & n(4) & = & 4 \\ n(5) & = & 6 & n(m) & = & 3^{m/3} \\ n(m+1) & = & \frac{4}{3} \times 3^{m/3} & n(m+2) & = & 2 \times 3^{m/3} \end{array}$$

Table 7 shows some comparison values between $n(m)$ and $n^{✔}(g)$ and $n^{✘}(g)$, for same-sized programs, based on Table 2. It is worth emphasising that $n(m)$ is the *highest possible* value: it is defined as the *maximum* over *all* possible programs. However, the maximal program is highly atypical. For example, considering all programs with seven statements, there are

28. A path of $P_1; P_2$ simply concatenates a path of P_1 with a path of P_2, hence the product; and a path of $P_1 + P_2$ is either a path of P_1 *or* a path of P_2, hence the addition.
29. Using the total number of nodes in the tree yields almost identical results.

a total of 8,448 possible programs. Of these 8448 programs, only 32 have 12 paths (the maximum). Figure 12 shows for each number of paths (1–12) how many programs have that many paths. The maximum of 12 is clearly not typical: indeed, the mean number of paths for a seven statement program is 4.379, and the median is 4. If we consider all programs with 9 statements, then there are 366,080 such programs, but only 16 have the maximal number of paths (which is 27). The average number of paths across all the programs is 5.95.

Overall, looking at Table 7, we conclude that the number of paths for BDI programs is much larger than even the (atypical) maximal number of paths for a procedural program of the same size. This supports the conclusion that BDI programs are *harder* to test than procedural programs.

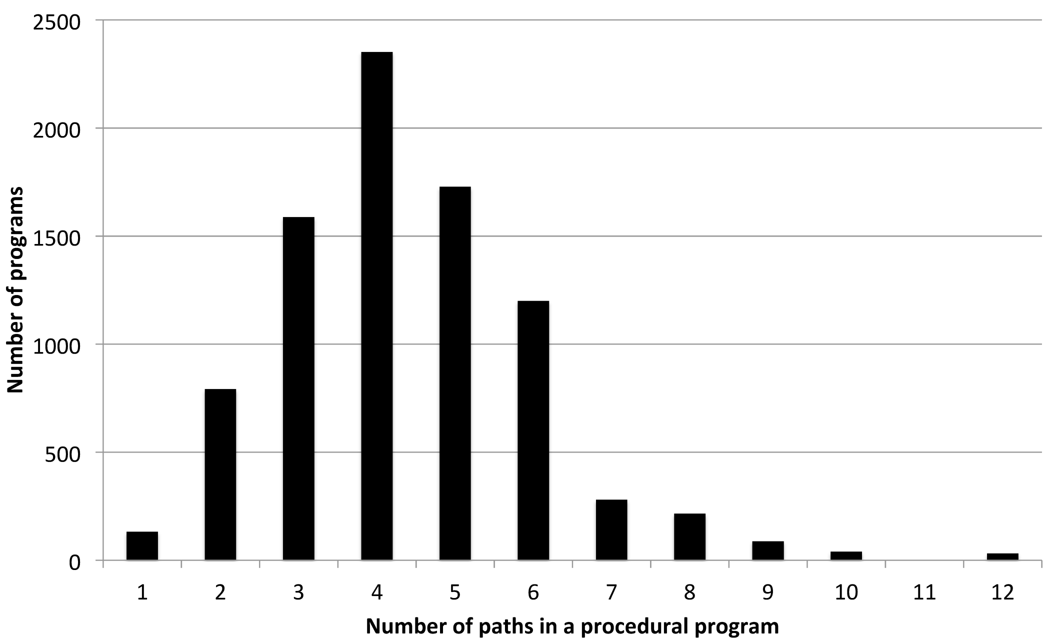


Figure 12: Profile of number of paths for all 7-statement programs

7. Conclusion

To summarise, our analysis has found that the space of possible behaviours for BDI agents is, indeed, large, both in an absolute sense, and in a relative sense (compared with procedural programs of the same size).

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. Flexibility is defined as the number of possible behaviours of an agent, which we have shown to be large. Robustness is defined as the ability of an agent to recover from failure. The analysis in Section 4.6 showed that the BDI failure recovery mechanism is effective at achieving a low rate of actual failure ($< 1\%$), even when each action has a reasonable chance of failing (5%).

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 6 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 1.1), an adequate test suite (using the "all paths" criterion for adequacy) requires at least as many tests as there are paths in the program being tested. If a program has, say, $\approx 10^{13}$ paths, then even a test suite with tens of thousands of tests is not just inadequate, but is hugely inadequate, since it only covers a tiny fraction of a percent of the number of paths.

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? Unfortunately, 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 (Dorigo & Stützle, 2004), 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. Consequently, it is difficult to draw conclusions about the correctness of a goal from the results of testing its sub-goals.

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 from a real application 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:

> 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 (Burmeister et al., 2008, p. 42)[30].

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 possible approaches for dealing with this issue of testability that could be recommended:

- **Keep BDI goal-plan trees shallow and sparse.** This 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 large number of paths does not encourage much optimism for this approach.

  Another, related, direction is to see whether patterns exist in the behaviour space. Since the failure recovery 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 in 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; but is not a viable testing strategy.

  Finally, a related direction is to try and be more intelligent about the selection of test cases, in order to gain more coverage from a given number of test cases. One approach for doing this, which has been recently described, is evolutionary testing (Nguyen, Miles, Perini, Tonella, Harman, & Luck, 2009a), in which genetic evolution is used to find good (i.e. challenging) test cases.

- **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 method[31]. Unfortunately, formal methods techniques are not yet applicable to industry-sized agent systems (we return to this below, in Section 7.1).

30. Burmeister et al. made the following observation: "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.
31. See the volume edited by Dastani et al. (2010) for a recent overview of the current state-of-the-art, including a chapter on the role of formal methods in the assurance of agents (Winikoff, 2010).

- **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 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 he/she will behave appropriately in all situations. Consequently, 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).

7.1 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 (van Riemsdijk, Dastani, & Winikoff, 2008). 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 extension of the analysis is to consider other criteria for test suite adequacy. In this paper we have used the "all paths" criterion, arguing why it is appropriate. We do recognize that "all paths" is actually quite a strong criterion—it subsumes many other criteria (Zhu et al., 1997, Figure 7). An alternative criterion that we could consider is "all edges", also known as "branch coverage" or "decision coverage". It requires that where there is a choice in the program, such as an "if" statement, then the tests in the test suite exercise all options, i.e. that all edges in the program graph be covered. The "all edges" criterion is weaker than "all paths" and is regarded as "the generally accepted minimum" (Jorgensen, 2002).

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 (Thangarajah, Winikoff, Padgham, & Fischer, 2002). 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 real applications), 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[32] (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 difficulties in assuring BDI agent systems through testing, we need to find other ways of assuring such systems.

An approach that has some promise is the automatic generation of test cases for agent systems (Nguyen, Perini, & Tonella, 2007; Zhang, Thangarajah, & Padgham, 2009). 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) (Dwyer, Hatcliff, Pasareanu, Robby, & Visser, 2007).

Another approach[33] that has attracted interest is model checking of agent systems (Wooldridge, Fisher, Huget, & Parsons, 2002; Bordini, Fisher, Pardavila, & Wooldridge, 2003; Raimondi & Lomuscio, 2007). 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 (Burch, Clarke, McMillan, Dill, & Hwang, 1992; Fix, Grumberg, Heyman, Heyman, & Schuster, 2005). 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 (2007) verify systems where agents are defined abstractly, i.e. not in terms of plans and goals. The MABLE agent programming language (Wooldridge et al., 2002) is actually an imperative language augmented with certain agent features, not a BDI language; and the work of Bordini et al. (2003) 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 (Dennis, Fisher, Webster, & Bordini, 2012).

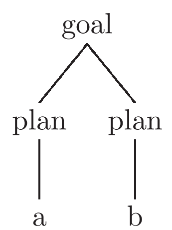
Acknowledgements

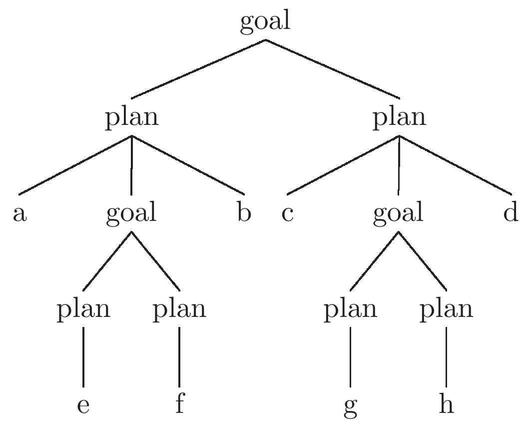
We would like to thank members of the Department of Information Science at the University of Otago for discussions relating to this paper. We would also like to thank Lin Padgham for her comments on a draft of this paper. Finally, we would like to thank the anonymous reviewers for their insightful comments which helped to improve this paper. Some of the work on this paper was done while Winikoff was on sabbatical from RMIT, visiting the University of Otago.

32. Generating all traces of the top level goal is not likely to be feasible.
33. 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.

Appendix 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`.





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[34]. As predicted by our formulae, there are four successful executions and two 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 e b ✘ c g d
a e b ✘ c g d ✘
a e b ✘ c g ✘ h d
a e b ✘ c g ✘ h d ✘
a e b ✘ c g ✘ h ✘
a e b ✘ c h d
a e b ✘ c h d ✘
a e b ✘ c h ✘ g d
a e b ✘ c h ✘ g d ✘
a e b ✘ c h ✘ g ✘
a e b ✘ c ✘
a e ✘ f b
a e ✘ f b ✘ c g d
a e ✘ f b ✘ c g d ✘
a e ✘ f b ✘ c g ✘ h d
a e ✘ f b ✘ c g ✘ h d ✘
a e ✘ f b ✘ c g ✘ h ✘
a e ✘ f b ✘ c h d
a e ✘ f b ✘ c h d ✘
a f b ✘ c h d
a f b ✘ c h d ✘
a f b ✘ c h ✘ g d
a f b ✘ c h ✘ g d ✘
a f b ✘ c h ✘ g ✘
a f b ✘ c ✘
a f ✘ e b
a f ✘ e b ✘ c g d
a f ✘ e b ✘ c g d ✘
a f ✘ e b ✘ c g ✘ h d
a f ✘ e b ✘ c g ✘ h d ✘
a f ✘ e b ✘ c g ✘ h ✘
a f ✘ e b ✘ c h d
a f ✘ e b ✘ c h d ✘
a f ✘ e b ✘ c h ✘ g d
a f ✘ e b ✘ c h ✘ g d ✘
a f ✘ e b ✘ c h ✘ g ✘
a f ✘ e b ✘ c ✘
a f ✘ e ✘ c g d
a f ✘ e ✘ c g d ✘
c g d ✘ a e b
c g d ✘ a e b ✘
c g d ✘ a e ✘ f b
c g d ✘ a e ✘ f b ✘
c g d ✘ a e ✘ f ✘
c g d ✘ a f b
c g d ✘ a f b ✘
c g d ✘ a f ✘ e b
c g d ✘ a f ✘ e b ✘
c g d ✘ a f ✘ e ✘
c g d ✘ a ✘
c g ✘ h d
c g ✘ h d ✘ a e b
c g ✘ h d ✘ a e b ✘
c g ✘ h d ✘ a e ✘ f b
c g ✘ h d ✘ a e ✘ f b ✘
c g ✘ h d ✘ a e ✘ f ✘
c g ✘ h d ✘ a f b
c g ✘ h d ✘ a f b ✘
c g ✘ h d ✘ a f ✘ e b
c h d ✘ a f b ✘
c h d ✘ a f ✘ e b
c h d ✘ a f ✘ e b ✘
c h d ✘ a f ✘ e ✘
c h d ✘ a ✘
c h ✘ g d
c h ✘ g d ✘ a e b
c h ✘ g d ✘ a e b ✘
c h ✘ g d ✘ a e ✘ f b
c h ✘ g d ✘ a e ✘ f b ✘
c h ✘ g d ✘ a e ✘ f ✘
c h ✘ g d ✘ a f b
c h ✘ g d ✘ a f b ✘
c h ✘ g d ✘ a f ✘ e b
c h ✘ g d ✘ a f ✘ e b ✘
c h ✘ g d ✘ a f ✘ e ✘
c h ✘ g d ✘ a ✘
c h ✘ g ✘ a e b
c h ✘ g ✘ a e b ✘
c h ✘ g ✘ a e ✘ f b

34. Note that the failure marker isn't counted when considering the length of the trace in Section 4.6.

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

Appendix B. Analysis of Recurrence Relations

This appendix contains details of the derivation in Section 4.4.

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

$$\begin{aligned} F(x) &= \sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)\frac{x^j}{j!} \qquad (24)\\ &\quad \text{(by definition of } f^{\checkmark}\text{)} \\ &= \sum_{j=0}^{\infty}\left(\sum_{i=0}^{j-1}\binom{j}{i} i!a^i(j-i)b\right)\frac{x^j}{j!} = \sum_{j=0}^{\infty}\left(\sum_{i=0}^{\infty}\binom{j}{i} i!a^i(j-i)b\right)\frac{x^j}{j!} \end{aligned}$$

On the right hand side above we have changed the upper limit of the inner sum to ∞ based on the generalised definition of $\binom{j}{i}$ as $j(j-1)(j-2)\ldots(j-i+1)/i!$, which is valid for all complex numbers j and non-zero integers i (Wilf, 1994) and gives $\binom{j}{i} = 0$ for $i > j$.

The right hand side has the form of a product of exponential generating functions (Wilf, 1994, Rule 3′, Section 2.3):

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

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

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

The left hand sum is $G(ax)$ where $G(y) = \sum_n y^n = \frac{1}{1-y}$ (Wilf, 1994, Equation 2.5.1)[35]. The right hand sum is equal to $bx\frac{d}{dx}\left(\sum \frac{x^n}{n!}\right)$ (Wilf, 1994, Rule 2′, Section 2.3) $= bx\frac{d}{dx}e^x$ (Wilf, 1994, 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 $\sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)\frac{x^j}{j!}$, which is $F(0) = 0$. To find a recurrence relation defining $f^{\checkmark}(j+1,a,b)$ we equate the original definition of $F(x)$ in Equation 24 with our closed form of this function, differentiate each side (to give us a power series with the $f^{\checkmark}(j,a,b)$ values shifted one position to the left), and multiply by the denominator of the closed form, giving us the following derivation.

$$\begin{aligned}
& (1-ax)\frac{d}{dx}\left(\sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)\frac{x^j}{j!}\right) = (1-ax)\frac{d}{dx}\left(\frac{bxe^x}{1-ax}\right) \\
\Longrightarrow\ & (1-ax)\sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)j\frac{x^{j-1}}{j!} = (1-ax)\left(\frac{b(x+1)e^x}{1-ax} + \frac{abxe^x}{(1-ax)^2}\right) \\
\Longrightarrow\ & \sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)j\frac{x^{j-1}}{j!} - \sum_{j=0}^{\infty} af^{\checkmark}(j,a,b)j\frac{x^j}{j!} = b(x+1)e^x + a\frac{bxe^x}{1-ax} \\
& \text{(recall that } f^{\checkmark}(0,a,b) = 0) \\
\Longrightarrow\ & \sum_{j=0}^{\infty} f^{\checkmark}(j+1,a,b)(j+1)\frac{x^j}{(j+1)!} - \sum_{j=0}^{\infty} ajf^{\checkmark}(j,a,b)\frac{x^j}{j!} \\
& \qquad = bxe^x + be^x + a\sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)\frac{x^j}{j!} \\
& \text{(recall that } bxe^x = \textstyle\sum_{j=0}^{\infty} jb\frac{x^j}{j!}, \text{ and } e^x = \sum_{j=0}^{\infty}\frac{x^j}{j!}) \\
& \qquad = b\sum_{j=0}^{\infty} j\frac{x^j}{j!} + b\sum_{j=0}^{\infty}\frac{x^j}{j!} + a\sum_{j=0}^{\infty} f^{\checkmark}(j,a,b)\frac{x^j}{j!}
\end{aligned}$$

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

$$\begin{aligned}
& f^{\checkmark}(j+1,a,b) - ajf^{\checkmark}(j,a,b) = bj + b + af^{\checkmark}(j,a,b) \\
\Longrightarrow\ & f^{\checkmark}(j+1,a,b) = b(j+1) + af^{\checkmark}(j,a,b) + ajf^{\checkmark}(j,a,b) \\
& \qquad = (j+1)(b + af^{\checkmark}(j,a,b))
\end{aligned} \tag{25}$$

35. Note that many of the operations performed on generating functions (and all those used in this paper) are valid without concern about convergence of the series. In combinatorics, generating functions are often treated not as analytic functions to be evaluated for specific variable values, but rather as formal (possibly infinite) algebraic objects, which have well defined operations such as addition and multiplication. The set of *formal power series* over a finite set of variables has the structure of a "ring" from abstract algebra, and in this ring there is no notion of function convergence and evaluation (Wilf, 1994, ch. 2).

Appendix C. Analysis of Recursive Goal-Plan Trees

This appendix contains detailed derivations relating to Section 4.7.

C.1 Derivation of $F^{✔}_{\infty}(goal(PM), x, y, b, \lambda)$

We can define $F^{✔}_{\infty}(goal(PM), x, y, b, \lambda)$ in terms of $n^{✔}_{\infty}$ in the usual way, noting the upper bound of λ to realise the length bound:

$$F^{✔}_{\infty}(goal(PM), x, y, b, \lambda) = \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n^{✔}_{\infty}(goal(PM), m, n, b) x^m y^n$$

where $n^{✔}_{\infty}(goal(PM), m, n, b)$ is as defined in Section 4.7. We also make use of the non-bounded version (which has only four arguments):

$$F^{✔}_{\infty}(goal(PM), x, y, b) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} n^{✔}_{\infty}(goal(PM), m, n, b) x^m y^n$$

We then can define $n^{✔}_{\infty}$ by counting successful traces:

$$\begin{aligned} & n^{✔}_{\infty}(goal(PM), m, n, b) \\ = & \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p) \sum_{\substack{m_1+m_2=m \\ n_1+n_2=n}} n^{✔}_{\infty}(p, m_1, n_1, b)\, n^{✘}_{\infty}(PM - \{p{:}1\}, m_2, n_2, b) \end{aligned}$$

where $n^{✘}_{\infty}(PM, m, n, b)$ is the number of unsuccessful paths using zero or more of the plans in the plan multiset PM (with respect to binding b) that have m actions, n of which are failed actions.

The inner sum considers all ways to partition the numbers of actions (m) and action failures (n) into those caused by a single plan of shape p and those caused by all the other plans. As in Section 4.6 (page 98) we use the identity $\sum_{m=0}^{\infty} \sum_{p+q=m} f(p, q) = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} f(p, q)$ to rewrite:

$$\begin{aligned} & F^{✔}_{\infty}(goal(PM), x, y, b) \\ = & \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p) \sum_{\substack{m_1+m_2=m \\ n_1+n_2=n}} n^{✔}_{\infty}(p, m_1, n_1, b)\, n^{✘}_{\infty}(PM - \{p{:}1\}, m_2, n_2, b) x^m y^n \\ = & \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p) \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{\substack{m_1+m_2=m \\ n_1+n_2=n}} n^{✔}_{\infty}(p, m_1, n_1, b)\, n^{✘}_{\infty}(PM - \{p{:}1\}, m_2, n_2, b) x^m y^n \end{aligned}$$

to give us:

$$\begin{aligned} & F^{✔}_{\infty}(goal(PM), x, y, b) \\ = & \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p) \\ & \quad \sum_{m_1=0}^{\infty} \sum_{m_2=0}^{\infty} \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} n^{✔}_{\infty}(p, m_1, n_1, b) x^{m_1} y^{n_1}\, n^{✘}_{\infty}(PM - \{p{:}1\}, m_2, n_2, b) x^{m_2} y^{n_2} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p) \\
&\qquad \sum_{m_1=0}^{\infty} \sum_{n_1=0}^{\infty} n_{\infty}^{\checkmark}(p, m_1, n_1, b) x^{m_1} y^{n_1} \sum_{m_2=0}^{\infty} \sum_{n_2=0}^{\infty} n_{\infty}^{\times}(PM - \{p{:}1\}, m_2, n_2, b) x^{m_2} y^{n_2} \\
&= \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p)\, F_{\infty}^{\checkmark}(p, x, y, b)\, F_{\infty}^{\times}(PM - \{p{:}1\}, x, y, b)
\end{aligned}
$$

where $F_{\infty}^{\times}(PM, x, y, b)$ is the generating function for $n_{\infty}^{\times}(PM, m, n, b)$. In Section C.3 we provide a definition of $F_{\infty}^{\times}(PM, x, y, b)$ in terms of an auxiliary function $G_{\infty}^{\times}$ (see Section C.4). We then introduce the bound on the length of paths λ giving:

$$
\begin{aligned}
&F_{\infty}^{\checkmark}(goal(PM), x, y, b, \lambda) \\
&\quad = \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p)\, (F_{\infty}^{\checkmark}(p, x, y, b) \overset{x \leq \lambda}{\times} F_{\infty}^{\times}(PM - \{p : 1\}, x, y, b)) \\
&\quad = \sum_{p \in \mathrm{set}(PM)} \chi_{PM}(p)\, (F_{\infty}^{\checkmark}(p, x, y, b, \lambda) \overset{x \leq \lambda}{\times} F_{\infty}^{\times}(PM - \{p : 1\}, x, y, b, \lambda))
\end{aligned}
$$

C.2 Derivation of $F_{\infty}^{\times}(goal(PM), x, y, b, \lambda)$

Similarly to the previous derivation, we define:

$$F_{\infty}^{\times}(goal(PM), x, y, b, \lambda) = \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n_{\infty}^{\times}(goal(PM), m, n, b) x^m y^n$$

To derive a recursive definition of $F_{\infty}^{\times}(goal(PM), \dots)$ in terms of the plans in PM, we first define a new function $n_{\infty}^{\times}(PM, m, n, o, b)$, which denotes the number of unsuccessful paths that use o of the plans in the multiset PM. We then have:

$$n_{\infty}^{\times}(goal(PM), m, n, b) = n_{\infty}^{\times}(PM, m, n, |PM|, b)$$

This states that for the goal to fail, all $|PM|$ plans in the multiset must be tried.

We define a generating function $G_{\infty}^{\times}(PM, x, y, z, b, \lambda)$ for $n_{\infty}^{\times}(PM, m, n, o, b)$ that is ordinary in x and y but exponential in z, i.e. the coefficients of $x^m y^n z^o / o!$ are the values of $n_{\infty}^{\times}(PM, m, n, o, b)$.

We now have:

$$F_{\infty}^{\times}(goal(PM), x, y, b, \lambda) = \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n_{\infty}^{\times}(PM, m, n, |PM|, b) x^m y^n$$

which we wish to rewrite in terms of $G_{\infty}^{\times}$. We do this by generalising the right hand side to sum over possible values for the number of plans used (o), followed by a restriction to select only values where $o = |PM|$:

$$
\begin{aligned}
& F^{\times}_{\infty}(goal(PM), x, y, b, \lambda) \\
&= \quad |PM|! \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} \frac{n^{\times}_{\infty}(PM, m, n, |PM|, b)\, z^{|PM|}}{|PM|!\, z^{|PM|}} x^m y^n \\
&= \quad |PM|! \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} \left(\frac{\sum_{o=0}^{\infty} n^{\times}_{\infty}(PM, m, n, o, b) z^o / o! \upharpoonright_{\text{power}(z)=|PM|}}{z^{|PM|}} \right) x^m y^n \\
&= \quad |PM|! \frac{\left(\sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} \sum_{o=0}^{\infty} n^{\times}_{\infty}(PM, m, n, o, b) x^m y^n z^o / o! \right) \upharpoonright_{\text{power}(z)=|PM|}}{z^{|PM|}}
\end{aligned}
$$

Since the nested sum is just the definition of $G^{\times}_{\infty}$ (see Section C.4), we can simplify this to:

$$
F^{\times}_{\infty}(goal(PM), x, y, b, \lambda) \quad = \quad |PM|! \; \frac{G^{\times}_{\infty}(PM, x, y, z, b, \lambda) \upharpoonright_{\text{power}(z)=|PM|}}{z^{|PM|}}
$$

In Section C.4 we derive a definition of $G^{\times}_{\infty}(PM, \ldots)$ in terms of $F^{\times}_{\infty}(p, \ldots)$ for each $p \in \text{set}(PM)$.

C.3 Definition of $F^{\times}_{\infty}(PM, x, y, b, \lambda)$

Recall that $n^{\times}_{\infty}(PM, m, n, b)$ is the number of unsuccessful paths using zero or more of the plans in the plan multiset PM (with respect to binding b) that have m actions, n of which are failed actions, and $F^{\times}_{\infty}(PM, x, y, b, \lambda)$ is its ordinary generating function.

First we consider the case when PM is empty. In this case, there is precisely one way to fail, and it generates a trace of length zero. Therefore, $F^{\times}_{\infty}(\{\}, x, y, b, \lambda) = 1x^0y^0 = 1$. For the case when PM is non-empty we can sum over the number of plans used during execution, which yields the following definition:

$$
n^{\times}_{\infty}(PM, m, n, b) = \sum_{o=0}^{|PM|} n^{\times}_{\infty}(PM, m, n, o, b)
$$

where $n^{\times}_{\infty}(PM, m, n, o, b)$ is, as before, the number of unsuccessful paths through the plan multiset PM, using o of the plans. Therefore, using the definition of $F^{\times}_{\infty}(PM, x, y, b, \lambda) = \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} n^{\times}_{\infty}(PM, m, n, o, b) x^m y^n$, we have:

$$
\begin{aligned}
& F^{\times}_{\infty}(PM, x, y, b, \lambda) \\
&= \quad \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} \sum_{o=0}^{|PM|} n^{\times}_{\infty}(PM, m, n, o, b) x^m y^n \\
& \quad\quad \text{(replace } n^{\times}_{\infty} \text{ by looking up the coefficient of the corresponding term in } G^{\times}_{\infty}, \\
& \quad\quad\ \text{the } o! \text{ accounts for the division by } o! \text{ in } G^{\times}_{\infty}\text{; we also reorder the summations)} \\
&= \quad \sum_{o=0}^{|PM|} \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} o! [x^m y^n z^o] G^{\times}_{\infty}(PM, x, y, z, b, \lambda) x^m y^n
\end{aligned}
$$

(we shift $o!$ outwards, and multiply by z^o/z^o)

$$= \sum_{o=0}^{|PM|} o! \frac{\sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} [x^m y^n z^o] G_{\infty}^{\text{✘}}(PM, x, y, z, b, \lambda) x^m y^n z^o}{z^o}$$

$$= \sum_{o=0}^{|PM|} o! \frac{G_{\infty}^{\text{✘}}(PM, x, y, z, b, \lambda) \restriction_{\mathrm{power}(z)=o}}{z^o}$$

C.4 Definition of $G_{\infty}^{\text{✘}}(PM, x, y, z, b, \lambda)$

We define $G_{\infty}^{\text{✘}}(PM, x, y, z, b, \lambda)$ as a generating function for $n_{\infty}^{\text{✘}}(PM, m, n, o, b)$ that is ordinary in x and y but exponential in z (hence the division by $o!$ below), with $\lambda(\geq 0)$ as the maximum allowed trace length:

$$G_{\infty}^{\text{✘}}(PM, x, y, z, b, \lambda) = \sum_{m=0}^{\lambda} \sum_{n=0}^{\infty} \sum_{o=0}^{\infty} n_{\infty}^{\text{✘}}(PM, m, n, o, b) \frac{x^m y^n z^o}{o!}$$

Recall that $n_{\infty}^{\text{✘}}(PM, m, n, o, b)$ denotes the number of unsuccessful paths that use o of the plans in the multiset PM. For an empty multiset of plans there is no successful execution, and there is a single unsuccessful execution with 0 actions, that uses 0 plans, hence:

$$n_{\infty}^{\text{✘}}(\{\}, m, n, o, b) = \begin{cases} 1 & \text{if } m = n = o = 0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore, $G_{\infty}^{\text{✘}}(\{\}, x, y, z, b, \lambda) = 1$. For non-empty multisets we must partition the actions in each trace, the action failures, and numbers of plans used, across the different plan bodies in the multiset, and also consider all ways that the plans of the various plan shapes can be interleaved to give an overall order for attempting plans:

$$\begin{aligned} & n_{\infty}^{\text{✘}}(\{p_1{:}c_1, \ldots, p_j{:}c_j\}, m, n, o, b) \\ & \quad = \sum_{\substack{m_1+\cdots+m_j=m \\ n_1+\cdots+n_j=n \\ o_1+\cdots+o_j=o}} \Phi \;\; n_{\infty}^{\text{✘}}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \cdots n_{\infty}^{\text{✘}}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \end{aligned}$$

where Φ is the multinomial coefficient $\binom{o}{o_1 \ldots o_j} = \frac{o!}{o_1! \ldots o_j!}$
Thus:

$G_{\infty}^{\text{✘}}(\{p_1{:}c_1, \ldots, p_j{:}c_j\}, x, y, z, b, \lambda)$
(by definition of $G_{\infty}^{\text{✘}}$, but using restriction, rather than a bounded sum on m,
and expanding $n_{\infty}^{\text{✘}}$ as above)

$$= \left(\sum_{m,n,o=0}^{\infty} \sum_{\substack{m_1+\cdots+m_j=m \\ n_1+\cdots+n_j=n \\ o_1+\cdots+o_j=o}} \binom{o}{o_1 \ldots o_j} n_{\infty}^{\text{✘}}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \right.$$

$$\left. \cdots n_{\infty}^{\text{✘}}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \frac{x^m y^n z^o}{o!} \right) \restriction_{\mathrm{power}(x) \leq \lambda}$$

$$= \left(\sum_{m,n,o=0}^{\infty} \sum_{\substack{m_1+\cdots+m_j=m \\ n_1+\cdots+n_j=n \\ o_1+\cdots+o_j=o}} \frac{o!}{o_1! \ldots o_j!} n^{\boldsymbol{\times}}_{\infty}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \right.$$

$$\left. \cdots n^{\boldsymbol{\times}}_{\infty}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \frac{x^m y^n z^o}{o!} \right) \upharpoonright_{\text{power}(x) \leq \lambda}$$

(cancelling $o!$ and distributing the $o_i!$ and the x^{m_i}, y^{n_i} and z^{o_i})

$$= \left(\sum_{m,n,o=0}^{\infty} \sum_{\substack{m_1+\cdots+m_j=m \\ n_1+\cdots+n_j=n \\ o_1+\cdots+o_j=o}} n^{\boldsymbol{\times}}_{\infty}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \frac{x^{m_1} y^{n_1} z^{o_1}}{o_1!} \right.$$

$$\left. \cdots n^{\boldsymbol{\times}}_{\infty}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \frac{x^{m_j} y^{n_j} z^{o_j}}{o_j!} \right) \upharpoonright_{\text{power}(x) \leq \lambda}$$

(replacing $\sum_{m=0}^{\infty} \sum_{m_1+m_2=m}$ with $\sum_{m_1=0}^{\infty} \sum_{m_2=0}^{\infty}$ and redistributing sums)

$$= \left(\left(\sum_{m_1,n_1,o_1=0}^{\infty} n^{\boldsymbol{\times}}_{\infty}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \frac{x^{m_1} y^{n_1} z^{o_1}}{o_1!} \right) \right.$$

$$\left. \cdots \left(\sum_{m_j,n_j,o_j=0}^{\infty} n^{\boldsymbol{\times}}_{\infty}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \frac{x^{m_j} y^{n_j} z^{o_j}}{o_j!} \right) \right) \upharpoonright_{\text{power}(x) \leq \lambda}$$

(replacing restriction $\upharpoonright$ with $\overset{x \leq \lambda}{\times}$)

$$= \left(\sum_{m_1,n_1,o_1=0}^{\infty} n^{\boldsymbol{\times}}_{\infty}(\{p_1{:}c_1\}, m_1, n_1, o_1, b) \frac{x^{m_1} y^{n_1} z^{o_1}}{o_1!} \right) \overset{x \leq \lambda}{\times}$$

$$\cdots \overset{x \leq \lambda}{\times} \left(\sum_{m_j,n_j,o_j=0}^{\infty} n^{\boldsymbol{\times}}_{\infty}(\{p_j{:}c_j\}, m_j, n_j, o_j, b) \frac{x^{m_j} y^{n_j} z^{o_j}}{o_j!} \right)$$

(by definition of $G^{\boldsymbol{\times}}_{\infty} \ldots$)

$$= G^{\boldsymbol{\times}}_{\infty}(\{p_1{:}c_1\}, x, y, z, b, \lambda) \overset{x \leq \lambda}{\times} \cdots \overset{x \leq \lambda}{\times} G^{\boldsymbol{\times}}_{\infty}(\{p_j{:}c_j\}, x, y, z, b, \lambda)$$

We now need to define $G^{\boldsymbol{\times}}_{\infty}(\{p_i{:}c_i\}, x, y, z, b, \lambda)$.

Consider $n^{\boldsymbol{\times}}_{\infty}(\{p{:}c\}, m, n, o, b)$. The simple cases are where $o = 0$: if we do not use any plans, then there is only a single unsuccessful path, which has no actions ($m = n = 0$). On the other hand, if $o > 0$ then we have $\binom{c}{o} = \frac{c!}{o!(c-o)!}$ ways of selecting o out of the c available copies of the plan p. These selected plans can be executed in $o!$ different orders. For each execution we sum over the possible distributions of actions (successful and unsuccessful) amongst the plans. This gives:

$$n^{\times}_{\infty}(\{p{:}c\}, m, n, o, b) = \begin{cases} \binom{c}{o} o! \displaystyle\sum_{\substack{m_1+\cdots+m_o=m \\ n_1+\cdots+n_o=n}} n^{\times}_{\infty}(p, m_1, n_1, b) \cdots n^{\times}_{\infty}(p, m_o, n_o, b) & \text{if } o > 0 \\ 1 \quad \text{if } m = n = o = 0 \\ 0 \quad \text{otherwise} \end{cases}$$

We therefore have the following definition of $G^{\times}_{\infty}(\{p{:}c\}, x, y, z, b, \lambda)$, where the initial 1 abbreviates $1x^0y^0z^0/0!$, i.e. the base case where $m = n = o = 1$, and the rest is from the definition of $G^{\times}_{\infty}$, expanding $n^{\times}_{\infty}$ using the above definition.

$$\begin{aligned}
& G^{\times}_{\infty}(\{p{:}c\}, x, y, z, b, \lambda) \\
&= 1 + \sum_{m=0}^{\lambda} \sum_{n=0,o=1}^{\infty} \binom{c}{o} o! \sum_{\substack{m_1+\cdots+m_o=m \\ n_1+\cdots+n_o=n}} n^{\times}_{\infty}(p, m_1, n_1, b) \cdots n^{\times}_{\infty}(p, m_o, n_o, b) \frac{x^m y^n z^o}{o!} \\
& \quad \text{(cancel } o!/o!\text{, rearrange sums and replace an upper bound of } \lambda \text{ on } m \\
& \quad \text{with a restriction)} \\
&= 1 + \sum_{o=1}^{\infty} \binom{c}{o} \left(\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{\substack{m_1+\cdots+m_o=m \\ n_1+\cdots+n_o=n}} n^{\times}_{\infty}(p, m_1, n_1, b) \cdots n^{\times}_{\infty}(p, m_o, n_o, b) x^m y^n \right) \restriction_{\mathrm{power}(x) \le \lambda} z^o \\
& \quad \text{(replacing } \sum_{m=0}^{\infty} \sum_{m_1+m_2=m} \text{ with } \sum_{m_1=0}^{\infty} \sum_{m_2=0}^{\infty} \text{ and redistributing sums)} \\
&= 1 + \sum_{o=1}^{\infty} \binom{c}{o} \\
& \quad \left(\left(\sum_{m_1=0}^{\infty} \sum_{n_1=0}^{\infty} n^{\times}_{\infty}(p, m_1, n_1, b) x^{m_1} y^{n_1} \right) \cdots \left(\sum_{m_o=0}^{\infty} \sum_{n_o=0}^{\infty} n^{\times}_{\infty}(p, m_o, n_o, b)\ x^{m_o} y^{n_o} \right) \right) \\
& \quad \restriction_{\mathrm{power}(x) \le \lambda} \ z^o \\
&= 1 + \sum_{o=1}^{\infty} \binom{c}{o} \left(\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} n^{\times}_{\infty}(p, m, n, b)\ x^m y^n \right)^o \restriction_{\mathrm{power}(x) \le \lambda} \ z^o \\
& \quad \text{(Replace } \sum_m \sum_n n^{\times}_{\infty}(p, m, n, b) x^m y^n \text{ with } F^{\times}_{\infty}(p, x, y, b, \lambda) \text{ as per its definition)} \\
&= 1 + \sum_{o=1}^{c} \binom{c}{o} F^{\times}_{\infty}(p, x, y, b, \lambda)^o \restriction_{\mathrm{power}(x) \le \lambda} \ z^o \\
&= 1 + \sum_{o=1}^{c} \binom{c}{o} F^{\times}_{\infty}(p, x, y, b, \lambda)^{o \restriction x \le \lambda} \ z^o
\end{aligned}$$

Appendix D. Analysis of Procedural Code Structures

We seek to derive an expression for the largest possible number of paths that a program of given size m can have, i.e. a definition of $n(m) = \max\{n(P) \ : \ |P| = m\}$. Recall that a program is either an (atomic) statement s which has a single path (i.e. $n(s) = 1$), a sequence of two programs $P_1; P_2$ where $n(P_1; P_1) = n(P_1) \times n(P_2)$, or a conditional $P_1 + P_2$ where $n(P_1 + P_2) = n(P_1) + n(P_2)$.

It is relatively easy to see by examining possible programs that for $m \leq 3$ we have $n(m) = m$. For instance, the largest number of paths for $m = 3$ is obtained by the program $s + s + s$. It is also easy to show that for $m = 4$ the largest number of paths possible is 4.

But what about larger values of m? We observe that for all $m > 4$ the program[36] with the largest number of paths follows a particular form. For $m = 5$ the program with the largest path can be written as $P_5 = (s + s + s); (s + s)$, and we have $n(P_5) = 3 \times 2$. More generally, we define S_2 to be $s + s$, and S_3 to be $s + s + s$, and we then have the following result, which shows that programs that have a maximal number of paths for their size, can be considered to be of a particular form.

Theorem D.1 *Any program of size i (for $i > 4$) that has the largest possible number of paths can be written as $P_i = P_i^1; P_i^2; \ldots; P_i^k$ where each of the P_i^j ($1 \leq j \leq k$) is either S_2 or S_3.*

Proof: *We establish this result by induction. We assume that it holds for all n such that $4 < n \leq m$, and then show that it must also hold for $m + 1$. So, let us assume that there is a program P_{m+1} which has a maximal number of paths, but is not in the form $P_{m+1}^1; P_{m+1}^2; \ldots; P_{m+1}^k$ where each P_{m+1}^j is either S_2 or S_3. There are two cases, depending on the structure of P_{m+1}. We consider each case in turn and show that in fact either (a) P_{m+1} can be rewritten to be in the desired form, preserving the number of paths and the program size; or (b) P_{m+1} cannot be maximal, since we can construct a program with size $m + 1$ that has a larger number of paths than P_{m+1}.*

Case 1: *P_{m+1} has the form $P_{m+1}^1; P_{m+1}^2; \ldots; P_{m+1}^k$ but at least one of the P_{m+1}^j is neither S_2 nor S_3. Let P_{m+1}^i be one of the sub-programs that is neither S_2 nor S_3. For convenience we define P^i as shorthand for P_{m+1}^i. Now, since P^i has size less than $m + 1$, the induction hypothesis applies[37], and so it can be written in the form $P_i^1; P_i^2; \ldots P_i^l$ where each P_i^j is either S_2 or S_3. It is easy to see that one can then rewrite P_{m+1} into the desired form by exploiting the associativity of ";", rewriting it as follows:*

$$\ldots P_{m+1}^{i-1}; (P_i^1; P_i^2; \ldots P_i^j); P_{m+1}^{i+1}; \ldots \implies \ldots P_{m+1}^{i-1}; P_i^1; P_i^2; \ldots P_i^j; P_{m+1}^{i+1}; \ldots$$

Applying this rewriting to all P_{m+1}^i that are not S_2 or S_3 yields a program that has size $m + 1$, the same number of paths as the original program, but that is in the desired form: a sequence of sub-programs, each of which is either S_2 or S_3. This shows that the result holds for $m + 1$, i.e. that a maximal-path program can be written in the desired form.

Case 2: *P_{m+1} does not have the form $P_{m+1}^1; P_{m+1}^2; \ldots; P_{m+1}^k$ for any k, which means that P_{m+1} must consist of a single conditional, i.e. $P_{m+1} = P_{m+1}^1 + \ldots + P_{m+1}^k$ for some $k > 1$.*

36. In fact there will be more than one maximal-path program, but they all have the same structure, modulo swapping the order of arguments to + and ;.
37. Or, if it has size 4, then it can be written as $S_2; S_2$ which has the maximal number of paths for a program of size 4 and meets the desired form.

Without loss of generality we can view P_{m+1} as being of the form $P^1_{m+1} + P^2_{m+1}$ (by viewing $P^1_{m+1} + \ldots + P^k_{m+1}$ as $(P^1_{m+1} + \ldots) + P^k_{m+1}$ if $k > 2$). We now consider the following sub-cases, depending on the values of $n(P^1_{m+1})$ and $n(P^2_{m+1})$.

Case 2a: *Both $n(P^1_{m+1})$ and $n(P^2_{m+1})$ are greater than 2. We can then show that P_{m+1} is not maximal. Consider the program $P'_{m+1} = P^1_{m+1}; P^2_{m+1}$ (i.e. where "+" is replaced by ";"). We know that $n(P^1_{m+1}; P^2_{m+1}) = n(P^1_{m+1}) \times n(P^2_{m+1})$. Without loss of generality, let's assume that $n(P^1_{m+1}) \leq n(P^2_{m+1})$. We then show that the original P_{m+1} has fewer paths than P'_{m+1}. The number of paths of P_{m+1} is $n(P_{m+1}) = n(P^1_{m+1}) + n(P^2_{m+1})$. Since $n(P^1_{m+1}) \leq n(P^2_{m+1})$, we have that $n(P_{m+1}) = n(P^1_{m+1}) + n(P^2_{m+1}) \leq n(P^2_{m+1}) + n(P^2_{m+1}) = 2 \times n(P^2_{m+1})$. Since $n(P^2_{m+1})$ and $n(P^1_{m+1})$ are both greater than 2, we then have that $2 \times n(P^2_{m+1}) < n(P^1_{m+1}) \times n(P^2_{m+1}) = n(P'_{m+1})$, i.e. that P'_{m+1} has more paths than P_{m+1}, and hence P_{m+1} is not maximal for $m + 1$.*

Case 2b: *At least one of $n(P^1_{m+1})$ and $n(P^2_{m+1})$ is not greater than 2. Without loss of generality, we assume that $n(P^1_{m+1}) \leq n(P^2_{m+1})$. There are then two cases: $n(P^1_{m+1})$ can be either 2 or 1.*

Sub-case 2b(i): *Let us consider first the case where $n(P^1_{m+1}) = 1$. Now the only program that has one path is a statement s, or a sequence of statements $s; s; \ldots; s$. Clearly the latter is not maximal since replacing it with $s + s + \ldots + s$ would result in a program of the same size but with more paths. So, therefore if P_{m+1} is maximal, then P^1_{m+1} must be just s, so $P_{m+1} = s + P^2_{m+1}$. Therefore P^2_{m+1} has size m. There are now two sub-cases: either m is still greater than 4, or $m = 4$. The second sub-case is simple: if m is 4 then we can show, by inspecting possible programs of size 4, that $n(4) = 4$, and we therefore have that $n(s + P^2_{m+1}) \leq 1 + 4 = 5$. However, we also know that $(s + s + s); (s + s)$ has size 5 but 6 paths, and hence in this sub-case P_{m+1} cannot have the maximal number of paths. In the first sub-case, where m is still greater than 4, the induction hypothesis applies and therefore P^2_{m+1} can be written in the desired form. We abbreviate P^2_{m+1} by P_2, and then have $P_{m+1} = s + (P^1_2; P^2_2; \ldots; P^j_2)$ where each P^i_2 is either S_2 or S_3. Consider now the variant program $P''_{m+1} = ((s + P^1_2); P^2_2; \ldots P^j_2)$, which clearly has the same size as P_{m+1}. We now show that P''_{m+1} has more paths than P_{m+1}: $n(P''_{m+1}) = ((1 + n(P^1_2)) \times n(P^2_2; \ldots; P^j_2)) = n(P^2_2; \ldots; P^j_2) + (n(P^1_2) \times n(P^2_2; \ldots; P^j_2))$. Now, $n(P_{m+1}) = 1 + (n(P^1_2) \times n(P^2_2; \ldots; P^j_2))$. In order to show that $n(P_{m+1}) < n(P''_{m+1})$ we just need to show that $1 < n(P^2_2; \ldots; P^j_2)$ which follows from the fact that there must be at least one P^i_2, and that, since each P^i_2 is either S_2 or S_3, it has size of at least 2.*

Sub-case 2b(ii): *We know that $n(P^1_{m+1}) = 2$ and that $2 \leq n(P^2_{m+1})$. Since $n(P^1_{m+1}) \leq n(P^2_{m+1})$ we have that $n(P^2_{m+1}) \geq 2$ and hence that $n(P_{m+1}) = 2 + n(P^2_{m+1}) \leq 2 \times n(P^2_{m+1}) = n(P'_{m+1})$. Now, if $n(P^2_{m+1})$ is strictly greater than 2 then we have that $n(P_{m+1})$ is strictly less than $n(P'_{m+1})$ and we have shown that P_{m+1} actually does not have a maximal number of paths. On the other hand, if $n(P^2_{m+1}) = 2$ then we have that $n(P_{m+1}) = 2 + n(P^2_{m+1}) = 2 + 2 = 4$. However, for values of m where this theorem applies, we know that $n(m) > 4$, and so we therefore have shown that in this sub-case P_{m+1} is not maximal for $m + 1$.*

We have shown that if we assume that P_{m+1} is maximal but does not have the structure specified, then in fact one can derive another program, also of size $m+1$, but which either does satisfy the desired structure, or has a larger number of paths than P_{m+1}, which contradicts our assumption that P_{m+1} is maximal. This establishes the desired property for P_{m+1}. By induction the result then applies for all $m > 4$, as desired. ∎

The previous result shows that when considering programs of a given size m that have the largest possible number of paths (denoted P_m), we can limit ourselves to considering programs that are of the form $P_1^m; P_2^m; \ldots; P_k^m$ where each P_i^m is either $s+s$ or $s+s+s$. We now derive a definition for $n(m)$. Firstly, we observe that, by inspecting cases:

$$\begin{aligned} n(m) &= m, \text{ if } m \leq 4 \\ n(5) &= 6 \\ n(6) &= 9 \end{aligned}$$

The two first cases have been discussed above. For the last case, there are only two programs which have the appropriate structure and have size 6: $S_2; S_2; S_2$ (with 8 paths) and $S_3; S_3$ (with 9 paths).

We now consider $m > 6$. Adding a statement to the program (i.e. going from m to $m+1$) in effect modifies P_m by adding an s to one of the P_i^m, which increments $n(P_i^m)$ by one. Since multiplication is commutative and associative, without loss of generality, we assume that we increment $n(P_k^m)$. We therefore have that $n(P_m) = n(P_1^m; \ldots; P_{k-1}^m) \times n(P_k^m)$ and that $n(P_{m+1}) = n(P_1^m; \ldots; P_{k-1}^m) \times (n(P_k^m) + 1)$. There are two cases:

Case 1: If all the P_i^m are S_3, then we have $n(P_m) = n(P_1^m; \ldots; P_{k-1}^m) \times 3$ and that therefore $n(P_{m+1}) = n(P_1^m; \ldots; P_{k-1}^m) \times 4 = n(P_m) \times \frac{4}{3}$. Note that in this case P_{m+1} can be written as $P_1^m; \ldots; P_{k-1}^m; S_2; S_2$.

Case 2: if some P_i^m are S_2 and some are S_3 then we observe that replacing a 2 with a 3 gives a greater increase to the number of paths than replacing a 3 with a 4, and hence (after possibly reordering the P_i^m so that $P_k^m = S_2$) we have $n(P_m) = n(P_1^m; \ldots; P_{k-1}^m) \times 2$ and that $n(P_{m+1}) = n(P_1^m; \ldots; P_{k-1}^m) \times 3 = n(P_m) \times \frac{3}{2}$.

We therefore have a recursive definition for $n(m)$ depending on the form of P_m. We next observe that in fact the form of P_m follows a simple cycle. We know that for $m = 6$, case 1 holds (as above, $P_6 = S_3; S_3$). We therefore have that P_7 can be written as $S_3; S_2; S_2$, hence P_8 can be written as $S_3; S_3; S_2$ or $S_3; S_2; S_3$, and hence P_9 can be written as $S_3; S_3; S_3$.

More generally, we can prove by induction that P_m can be written as $P_1^m; \ldots; P_k^m$ where the following holds: (a) if m is a multiple of 3, then all of the P_i^m are S_3; and (b) if m is one more than a multiple of 3, then exactly two of the P_i^m are S_2 and the rest are S_3; and (c) if m is two more than a multiple of 3, then exactly one of the P_i^m is S_2 and the rest are S_3. This gives us the following recursive definition, where $m \geq 6$ is a multiple of 3:

$$\begin{aligned} n(m+1) &= n(m) \times \frac{4}{3} \\ n(m+2) &= n(m+1) \times \frac{3}{2} \end{aligned}$$

$$n(m+3) \quad = \quad n(m+2) \times \frac{3}{2}$$

Which can be simplified to:

$$\begin{aligned}
n(m+1) &= n(m) \times \frac{4}{3} \\
n(m+2) &= n(m) \times \frac{3 \times 4}{2 \times 3} \;=\; 2 \times n(m) \\
n(m+3) &= n(m) \times \frac{3 \times 3 \times 4}{2 \times 2 \times 3} \;=\; 3 \times n(m)
\end{aligned}$$

We can easily derive a non-recursive definition by focusing on the last case and observing that as $n(6) = 9 = 3^2$ and $n(m+3) = 3 \times n(m)$ (for $m \geq 6$ being a multiple of 3), then we have that $n(m) = 3^{m/3}$. We can substitute this in the definition above to obtain the following complete definition for $n(m)$, where $m \geq 6$ is a multiple of 3:

$$\begin{aligned}
n(1) &= 1 \\
n(2) &= 2 \\
n(3) &= 3 \\
n(4) &= 4 \\
n(5) &= 6 \\
n(m) &= 3^{m/3} \\
n(m+1) &= \frac{4}{3} \times 3^{m/3} \\
n(m+2) &= 2 \times 3^{m/3}
\end{aligned}$$

References

Benfield, S. S., Hendrickson, J., & Galanti, D. (2006). Making a strong business case for multiagent technology. In Stone, P., & Weiss, G. (Eds.), *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 10–15. ACM Press.

Bordini, R. H., Fisher, M., Pardavila, C., & Wooldridge, M. (2003). Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 409–416. ACM Press.

Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley.

Bratman, M. E., Israel, D. J., & Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, *4*, 349–355.

Bratman, M. E. (1987). *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

Burch, J., Clarke, E., McMillan, K., Dill, D., & Hwang, J. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, *98*(2), 142–170.

Burmeister, B., Arnold, M., Copaciu, F., & Rimassa, G. (2008). BDI-agents for agile goal-oriented business processes. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS) [Industry Track]*, pp. 37–44. IFAAMAS.

Busetta, P., Rönnquist, R., Hodgson, A., & Lucas, A. (1999). JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News (2).

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

Dastani, M., Hindriks, K. V., & Meyer, J.-J. C. (Eds.). (2010). *Specification and Verification of Multi-agent systems*. Springer, Berlin/Heidelberg.

de Silva, L., & Padgham, L. (2004). A comparison of BDI based real-time reasoning and HTN based planning. In Webb, G., & Yu, X. (Eds.), *AI 2004: Advances in Artificial Intelligence*, Vol. 3339 of *Lecture Notes in Computer Science*, pp. 1167–1173. Springer, Berlin/Heidelberg.

Dennis, L. A., Fisher, M., Webster, M. P., & Bordini, R. H. (2012). Model checking agent programming languages. *Automated Software Engineering*, *19*(1), 3–63.

d'Inverno, M., Kinny, D., Luck, M., & Wooldridge, M. (1998). A formal specification of dMARS. In Singh, M., Rao, A., & Wooldridge, M. (Eds.), *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, Vol. 1365 of *Lecture Notes in Artificial Intelligence*, pp. 155–176, Berlin/Heidelberg. Springer.

Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization*. MIT Press.

Dwyer, M. B., Hatcliff, J., Pasareanu, C., Robby, & Visser, W. (2007). Formal software analysis: Emerging trends in software model checking. In *Future of Software Engineering 2007*, pp. 120–136, Los Alamitos, CA. IEEE Computer Society.

Ekinci, E. E., Tiryaki, A. M., Çetin, Ö., & Dikenelli, O. (2009). Goal-oriented agent testing revisited. In Luck, M., & Gomez-Sanz, J. J. (Eds.), *Agent-Oriented Software Engineering IX*, Vol. 5386 of *Lecture Notes in Computer Science*, pp. 173–186, Berlin/Heidelberg. Springer.

Erol, K., Hendler, J., & Nau, D. (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, *18*(1), 69–93.

Erol, K., Hendler, J. A., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pp. 1123–1128. AAAI Press.

Fix, L., Grumberg, O., Heyman, A., Heyman, T., & Schuster, A. (2005). Verifying very large industrial circuits using 100 processes and beyond. In Peled, D., & Tsay, Y.-K. (Eds.), *Automated Technology for Verification and Analysis*, Vol. 3707 of *Lecture Notes in Computer Science*, pp. 11–25, Berlin/Heidelberg. Springer.

Georgeff, M. P., & Lansky, A. L. (1986). Procedural knowledge. *Proceedings of the IEEE, Special Issue on Knowledge Representation*, *74*(10), 1383–1398.

Gomez-Sanz, J. J., Botía, J., Serrano, E., & Pavón, J. (2009). Testing and debugging of MAS interactions with INGENIAS. In Luck, M., & Gomez-Sanz, J. J. (Eds.), *Agent-Oriented Software Engineering IX*, Vol. 5386 of *Lecture Notes in Computer Science*, pp. 199–212, Berlin/Heidelberg. Springer.

Huber, M. J. (1999). JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pp. 236–243. ACM Press.

Ingrand, F. F., Georgeff, M. P., & Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert*, *7*(6), 33–44.

Jorgensen, P. (2002). *Software Testing: A Craftsman's Approach* (Second edition). CRC Press.

Lee, J., Huber, M. J., Kenny, P. G., & Durfee, E. H. (1994). 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. American Institute of Aeronautics and Astronautics.

Mathur, A. P. (2008). *Foundations of Software Testing*. Pearson.

Miller, J. C., & Maloney, C. J. (1963). Systematic mistake analysis of digital computer programs. *Communications of the ACM*, *6*(2), 58–63.

Morley, D., & Myers, K. (2004). The SPARK agent framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 714–721, New York. ACM.

Munroe, S., Miller, T., Belecheanu, R., Pechoucek, M., McBurney, P., & Luck, M. (2006). Crossing the agent technology chasm: Experiences and challenges in commercial applications of agents. *Knowledge Engineering Review*, *21*(4), 345–392.

Naish, L. (2007). Resource-oriented deadlock analysis. In Dahl, V., & Niemelä, I. (Eds.), *Proceedings of the 23rd International Conference on Logic Programming*, Vol. 4670 of *Lecture Notes in Computer Science*, pp. 302–316. Springer, Berlin/Heidelberg.

Nguyen, C., Miles, S., Perini, A., Tonella, P., Harman, M., & Luck, M. (2009a). Evolutionary testing of autonomous software agents. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 521–528. IFAAMAS.

Nguyen, C. D., Perini, A., & Tonella, P. (2009b). Experimental evaluation of ontology-based test generation for multi-agent systems. In Luck, M., & Gomez-Sanz, J. J. (Eds.), *Agent-Oriented Software Engineering IX*, Vol. 5386 of *Lecture Notes in Computer Science*, pp. 187–198, Berlin/Heidelberg. Springer.

Nguyen, C. D., Perini, A., & Tonella, P. (2007). Automated continuous testing of multi-agent systems. In *Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS)*.

Padgham, L., & Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons.

Paolucci, M., Shehory, O., Sycara, K. P., Kalp, D., & Pannu, A. (2000). A planning component for RETSINA agents. In Jennings, N. R., & Lespérance, Y. (Eds.), *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Vol. 1757 of *Lecture Notes in Computer Science*, pp. 147–161, Berlin/Heidelberg. Springer.

Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In Bordini, R. H., Dastani, M., Dix, J., & El Fallah Seghrouchni, A. (Eds.), *Multi-Agent Programming: Languages, Platforms and Applications*, chap. 6, pp. 149–174. Springer.

Raimondi, F., & Lomuscio, A. (2007). Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *J. Applied Logic*, *5*(2), 235–251.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W. V., & Perrame, J. (Eds.), *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, Vol. 1038 of *Lecture Notes in Artificial Intelligence*, pp. 42–55, Berlin/Heidelberg. Springer.

Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., & Sandewall, E. (Eds.), *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pp. 473–484. Morgan Kaufmann.

Sardina, S., & Padgham, L. (2011). A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, *23*(1), 18–70.

Shaw, P., Farwer, B., & Bordini, R. (2008). 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.

Sloane, N. J. A. (2007). The on-line encyclopedia of integer sequences. http://www.research.att.com/~njas/sequences/.

Thangarajah, J., Winikoff, M., Padgham, L., & Fischer, K. (2002). Avoiding resource conflicts in intelligent agents. In van Harmelen, F. (Ed.), *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI)*, pp. 18–22. IOS Press.

van Riemsdijk, M. B., Dastani, M., & Winikoff, M. (2008). Goals in agent systems: A unifying framework. In *Proceedings of the Seventh Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 713–720. IFAAMAS.

Wilf, H. S. (1994). *generatingfunctionology* (Second edition). Academic Press Inc., Boston, MA. http://www.math.upenn.edu/~wilf/gfology2.pdf.

Winikoff, M. (2010). Assurance of Agent Systems: What Role should Formal Verification play?. In Dastani, M., Hindriks, K. V., & Meyer, J.-J. C. (Eds.), *Specification and Verification of Multi-agent systems*, chap. 12, pp. 353–383. Springer, Berlin/Heidelberg.

Winikoff, M., Padgham, L., Harland, J., & Thangarajah, J. (2002). 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, Toulouse, France. Morgan Kaufmann.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley & Sons, Chichester, England.

Wooldridge, M., Fisher, M., Huget, M.-P., & Parsons, S. (2002). 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.

Zhang, Z., Thangarajah, J., & Padgham, L. (2009). Model based testing for agent systems. In Filipe, J., Shishkov, B., Helfert, M., & Maciaszek, L. (Eds.), *Software and Data Technologies*, Vol. 22 of *Communications in Computer and Information Science*, pp. 399–413, Berlin/Heidelberg. Springer.

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys*, *29*(4), 366–427.