TPLP **25** (5): 867–895, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068425100082 First published online 18 August 2025

Impact and Performance of Randomized Test Generation Using Prolog[*†]

MARCUS GELDERIE, MAXIMILIAN LUFF and MAXIMILIAN PELTZER
Aalen University of Applied Sciences Aalen, Germany
(*e-mails:* maximilian.luff@hs-aalen.de, marcus.gelderie@hs-aalen.de, maximilian.peltzer@hs-aalen.de)

submitted 15 January 2025; revised 23 June 2025; accepted 21 July 2025

Abstract

We study randomized generation of sequences of test inputs to a system using Prolog. Prolog is a natural fit to generate test sequences that have complex logical interdependent structure. To counter the problems posed by a large (or infinite) set of possible tests, randomization is a natural choice. We study the impact that randomization in conjunction with SLD resolution have on the test performance. To this end, this paper proposes two strategies to add randomization to a test-generating program. One strategy works on top of standard Prolog semantics, whereas the other alters the SLD selection function. We analyze the mean time to reach a test case and the mean number of generated test cases in the framework of Markov chains. Finally, we provide an additional empirical evaluation and comparison between both approaches.

KEYWORDS: software testing, randomization, Prolog

1 Introduction

The need for software testing is well established. The idea to auto-generate tests is a constant theme in the field of software testing, stretching back many decades (e.g. Miller and Melton 1975; Pesch *et al*. 1985; Ince 1987; Meyer *et al*. 2007). Automatically generating software tests can be done in a number of ways, depending on the specific test-goal: In the past, tests have been generated from UML specifications (Kim *et al*. 1999), based on natural language (Xu et al. 2022), and, more recently, using large-language models (Gu 2023; Siddiq *et al*. 2024). But tests have also been generated according to formal or semi-formal specifications (Zeng *et al*. 2002; Dewey et al. 2014). Particularly when formal methods are used, one often has to deal with a very large, even infinite,

* This work was created as part of a project funded by the German Federal Ministry of Education and Research under grant number 16KIS1913K. The authors are responsible for the content of this publication.
† This paper is an extended version of Gelderie *et al*. (2024), which was published in the 34th International Symposium on Logic-Based Program Synthesis and Transformation. We, thank the PC chairs Juliana Bowles and Harald Søndergaard.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

number of test cases. Exploring such a large set of tests in a randomized fashion is a natural approach and has been used extensively in various different ways and contexts for a long time (see e.g. Duran and Ntafos 1984; Miller *et al*. 1990; Miller *et al*. 1990; Ramler *et al*. 2012; Ramler *et al*. 2012; Casso *et al*. 2019).

Prolog is a natural fit to generate test cases that follow a logical pattern (as opposed to unstructured testing, as is done, e.g.,, in many forms of Fuzzing (Miller *et al*. 1990). Generating test cases using Prolog has been studied in the past (Pesch *et al*. 1985; Hoffman and Strooper 1991; Denney 1991; Casso *et al*. 2019). It has been applied to software-testing in general, but also to specialized areas, such as security testing (Dewey *et al*. 2014; Zech *et al*. 2019). Some approaches also use randomization to explore the space of test cases (Casso *et al*. 2019). Randomization solves some of the problems inherent in the SLD resolution algorithm – particularly the fact that it is not complete when the resolution works in a depth-first manner. It may also yield a more diverse set of test cases, because it permits exploring distant parts of an infinite SLD tree. Randomization seems to be a logical fit in the context of test-case generation using Prolog.

In the light of its apparent utility, it is natural to study randomization itself and its properties. What are the possible strategies to implement randomized search strategies for test cases in Prolog running on current state-of-the-art implementations? What is the probability of hitting a particular test case, and how long will it take? To our surprise, we only found very few papers dealing with the properties of randomization itself (see also *Related Work* below).

In this paper, we study randomized test-case generation using Prolog. Our main contributions are threefold: (i) We propose strategies to implement randomized search in both unmodified Prolog runtimes, and via specific modifications to the usual SLD implementations. (ii) We show how adding randomness naturally turns the SLD resolution into an infinite discrete-time Markov chain and propose to use this framework to study the runtime-effects. We do this for our proposed scheme and give tight asymptotic bounds on the expected time to hit a particular test case. (iii) Finally, we study the effect that various Prolog implementations have on the efficiency of randomizing test-case generation.

We present two ways of adding randomization to Prolog programs. The first way works without altering the semantics of standard Prolog and thus works on existing implementations. It works by adding a predicate, called a *guard*, that randomly fails to every rule. Crucially, failure is determined by an independent event for every successive call to the same rule. We refer to this strategy as the *guard approach*. In a second strategy, we propose a modification to the resolution algorithm: Given a goal and a set of matching rules, drop an indeterminate number of rules from the set and permute the remaining ones. Again, we do this in an independent fashion every time a goal is resolved with the input program. This second modification is reminiscent of that proposed in Casso *et al*. (2019), but differs in that it also *drops* a random number of rules from the set. This, in effect, prevents an infinite recursion with probability 1. We refer to the second strategy as the *drop-and-shuffle approach*.

In the following we study the effects on randomizing the resolution in this way. We give a detailed description of the resulting Markov chain and analyze its probability structure. We show that, provided the parameters are chosen appropriately, the number of test cases

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

produced is finite and given by a simple equation in terms of the selected probabilities. This is true for both approaches to randomization. We also show that, if we repeat the initial query infinitely many times, we will reach each test case after a finite number of steps on average. This *hitting time* is a well-known concept in the study of Markov chains. We again give a closed formula representation and accompanying asymptotic bound in the depth of the given test case in the SLD resolution tree. Again, this is done for both approaches, though the drop-and shuffle approach permits for a narrower set of parameters to ensure the computed quantities are finite.

Finally, we study the randomization procedures from an empirical perspective and provide comparisons between the two aforementioned approaches. We implement the guard approach to randomization in SWI-Prolog (Wielemaker *et al*. 2012) and the drop-and-shuffle approach in Go-Prolog ichiban/Prolog (2024). We chose Go-Prolog for its accessible and simple code-base, which lends itself to experimental modifications. We then compare the number of test cases produced before a specific test-goal is seen and the number of iterations that were required to do so.

1.1 Related work

Some early works on test-case generation using Prolog are Pesch *et al*. (1985), Bougé *et al*. (1985),Hoffman and Strooper (1991) and Denney (1991). Automated test-case generation in Prolog was described by Pesch *et al*. (1985). The authors state how to test individual syscalls with logic programming. The used specifications state a set of pre-conditions then the actual invocation of the respective syscall and afterwards what the expected post-conditions are. This paper demonstrates that test-case generation using Prolog is very beneficial to test systems in a structured manner. This problem domain does not deal with any problems of recursion since the authors only test input sequences of length one. This means that this paper does not deal with recursion problems that are witnessed for many other test scenarios.

Another approach showcasing Prolog's capabilities used in test case generation was shown by Hoffman and Strooper (1991). The authors automated the generation procedure of tests for modules written in C with Prolog.

Bougé *et al*. (1985) start the testing procedure with the definition of a Σ-algebra and respective axioms. The aim of this testing procedure is based on the regularity and uniformity testing hypothesis. Prolog is used to generate test cases and to partition the test cases into test classes following the uniformity hypothesis. The authors also recognize the problems of recursion in Prolog test case generation and apply different search strategies to solve them. Since the paper enforces a length limit on the generated solution, it will not find any test case that exceeds that length.

Denney (1991) also researches test-case generation based on specifications written in Prolog. In his paper, he implements a meta-interpreter in Prolog to be able to track which rules, generated from the specification, were already applied. This is done by constructing a finite automaton. Each arc between states corresponds with respective rules in the Prolog database. Final states in this automaton are test cases produced in the test-case generation process. With this solution, he addresses the problems of recursion, evaluable predicates, and ordering, which are challenging aspects of test-case

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

generation using Prolog. However, the recursion problem is only addressed heuristically, which means that a user has to specify a threshold of how often an arc can be traversed during the execution of test case generator. We argue that the estimation of the threshold is an error-prone task and, if not set correctly, could miss important test cases.

Gorlick *et al.* (1990) also introduce a methodology for formal specifications. For this task, they use constraint logic programming to describe the system under test's behavior. With this approach, the authors also recognize that they both have a test oracle and a test case generator at the same time. One challenge the authors addressed is, yet again, the recursion problem. To solve this challenge they used a randomization approach. This feature enables the proposed framework to pick probabilistically from the predicates. However, they do not provide any statements about test case duplication or infinite looping.

Casso *et al.* (2019) approach assertion-based testing of Prolog programs with random search rules. They rely on the `Ciao` model and its capabilities to specify pre- and post-conditions for static analysis and the runtime checker. Further, the authors develop a test-case generator based on these conditions. For randomizing the test case search, Casso et. al. use a selection function that randomly chooses clauses to be resolved. The authors do not study the randomization itself, nor its properties. We will revisit this paper and its randomization strategy in section 3, where we will also explain the differences from our approach in more detail.

Prolog was also used in security testing. For web applications, Zech *et al.* (2013, 2019) first build an expert system to filter test cases according to some attack pattern and later apply this risk analysis to filter test cases in the generation process. Since the paper, yet again, only addresses single input sequences, it effectively circumvents the problem of recursion. Prolog was also used in Fuzzing by Dewey *et al.* (2014) to use CLP in order to produce fuzzing inputs to compilers.

2 Preliminaries

Given a (usually finite) set Σ of elements, we write Σ^* for the set of all finite length sequences $w_1 \cdots w_l$ with $w_i \in \Sigma$ and $l \in \mathbb{N}_0 = \{0, 1, 2, 3, \ldots\} = \mathbb{N} \cup \{0\}$. The empty sequence is denoted by ε. We write $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $\Sigma = \{x\}$ is a singleton, we write x^* or x^+ instead of $\{x\}^*$. Concatenation is denoted by $(u_1 \cdots u_l) \cdot (v_1 \cdots v_r) = u_1 \cdots u_l v_1 \cdots v_r$. We write $|w| = |w_1 \cdots w_l| = l \in \mathbb{N}_0$.

We use the theory of *Markov chains*. For a detailed introduction and proofs of the following claims, the reader is referred to standard literature on the subject, for example, Norris (1998). We revisit the concepts, notation, and central results from the theory of Markov chains that we will use throughout this paper for convenience.

We consider a countable set $\mathcal{S}$ of *states*, a mapping $p\colon \mathcal{S} \times \mathcal{S} \to [0, 1]$ that assigns *transition probabilities* to pairs of states with the property that for all $s \in \mathcal{S}$ it holds that $\sum_{s' \in \mathcal{S}} p(s, s') = 1$, and an *initial state*[1] $\mathsf{Init} \in \mathcal{S}$. Let $(X_n)_{n \in \mathbb{N}_0}$ be an infinite sequence of

[1] In the literature one usually considers *initial distributions* to model uncertainty about the initial state. We do not need this capability in the present paper and consider initial distributions whose support is a single state of probability 1.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

random variables $X_n \in \mathcal{S}$. The tuple $(\mathcal{S}, (X_n)_{n\in\mathbb{N}_0}, p, \mathsf{Init})$ is a *Markov chain*, if $\Pr[X_0 = \mathsf{Init}] = 1$ and for all $n \in \mathbb{N}$ and all $s_1, \ldots, s_n \in \mathcal{S}$:

$$\Pr[X_n = s_n \mid X_0 = s_0 = \mathsf{Init}, \ldots, X_{n-1} = s_{n-1}] \\ = \Pr[X_n = s_n \mid X_{n-1} = s_{n-1}] = p(s_{n-1}, s_n)$$

For two states s, s' we write $s \rightsquigarrow s'$, if $\Pr[X_n = s' \text{ for some } n] > 0$ in the Markov chain $(\mathcal{S}, (X_n)_{n\in\mathbb{N}}, p, s)$. Intuitively, there is a way to get from s to s'. A set $A \subseteq \mathcal{S}$ is *absorbing*, if for every $s \in A$ and every $s' \in \mathcal{S}$ with $s \rightsquigarrow s'$ it holds that $s' \in A$. If $A = \{s\}$ is a singleton, the state s is said to be absorbing. If any two states are reachable from one-another ($s \rightsquigarrow s'$ for any $s, s' \in \mathcal{S}$), the Markov chain is *irreducible*.

Let $A \subseteq \mathcal{S}$ be a non-empty set of states and let $H^A = \inf\{n \in \mathbb{N}_0 \mid X_n \in A\} \in \mathbb{N}_0 \cup \{\infty\}$ denote the random variable such that $X_H \in A$ visits A for the first time. H^A is the *hitting time* of A. Then conditioned on $H^A < \infty$ and $X_H = s$, the sequence $(X_{H+n})_{n\in\mathbb{N}_0}$ is a Markov chain with initial state s and is independent of $X_0, \ldots, X_H$. This is called the *strong Markov property*. It is sometimes useful to consider the hitting times for initial states other than Init. Write H^A_s for the hitting time of A with starting state s.

The expected value $h^A \stackrel{\text{def}}{=} \mathsf{E}[H^A]$ is known as the *mean hitting time*. Given any state $s \in \mathcal{S}$, we define $h^A_s = \mathsf{E}[H^A_s]$ for the mean hitting time of A from initial state s. The mean hitting times are then the unique minimal positive solution to the equations

$$\begin{array}{ll} h^A_s = 0 & \text{if } s \in A \\ h^A_s = 1 + \sum_{s'\in\mathcal{S}} p(s, s') h^A_{s'} & \text{if } s \notin A \end{array} \tag{1}$$

A state $s \in \mathcal{S}$ is *recurrent*, if $\Pr[\sum_{n=0}^{\infty} \mathbb{1}_{X_n = s} = \infty] = 1$ (where $\mathbb{1}_A$ is the indicator random variable for event A). Otherwise, it is *transient*. It can be shown that a state is recurrent iff $\Pr[X_m = s \text{ for some } m \geq 1] = 1$ in the chain $(\mathcal{S}, (X_n)_{n\in\mathbb{N}_0}, p, s)$ (the probability of returning s, once visited, is 1). One can show that if a Markov chain is irreducible and contains one recurrent state, then all states are recurrent. In the case we call the chain itself *recurrent* (or *transient*).

3 Randomized test generation with prolog

In this paper, we view a test as a sequence of inputs to a system. For example, given a web-application with a REST-interface, we could think of a test as a sequence of HTTP-Requests using various methods (`GET`, `POST` and so forth) against different API-endpoints (e.g. `/login`, `/items/{USERID}/list`). Since our focus is on randomization, we do not explicitly model a concept of "valid" test cases. We also do not model the *test-oracle* which determines the success or failure of the test (e.g. "requests are processed in < 700 ms").

At a very abstract level, such a sequence of test inputs could be generated with the Prolog program shown in listing 1. All valid substitutions for `X` in the query `t(X)` are input sequences to our fictional system. Since this program will only ever output test sequences of the type `[command1, command1, command1, ...]`, a straightforward approach is to add *guard clauses* of the form shown in listing 2. Note that the symbols `p_cont`, `p_1`,...are meant to represent float constants between 0 and 1, and can be adjusted as needed.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

```
t([]).
t([H|T]) :- command(H), t(T).
command(X) :- command1(X); /* ... */ ; commandr(X).
command1(X) :- /* ... */ .
% ...
```

Listing. 1. A program generating randomized sequences of test inputs.

```
guard_t :- random(X), X < p_cont.
guard_1 :- random(X), X < p_1.
% ...
t([H|T]) :- guard_t, command(H), t(T)
command1(X) :- guard_1, /* ... */ .
% ...
```

Listing. 2. Guard clauses.

In effect, some sub-trees of the SLD-tree are then randomly left unexplored. We refer to this as the *guard approach*.

Another, superficially similar strategy was proposed by Casso *et al*. (2019). Their randomization is presented as a modification to the Prolog interpreter; equivalently, it can be implemented using meta-predicates. Essentially, Casso et. al. shuffle the list of input clauses whose head unifies with the current goal, instead of iterating over it in the usual left-to-right fashion. They do not drop rules. The termination of the program is instead enforced via depth-control. It is thus not difficult to see that the random approach itself merely alters the order of test cases, but not their number. As such, the questions concerning the number of test cases (that we study here) do not make sense for their approach.

However, one can augment the shuffling approach due to Casso et al. by *additionally* dropping several items from the set of unifying rules prior to shuffling. We do this with an independent Bernoulli trial for each rule (i.e. the number of dropped rules follows a Binomial). The resulting algorithm shares many properties with our scheme above (in particular, the results from the next section apply). We refer to this approach as the *drop-and-shuffle* strategy. We proceed to study both approaches below.

3.1 Guard strategy

3.1.1 Number of generated tests

The program $\mathcal{P}$ shown in listings 1 and 2 gives rise to a probabilistic number of test cases. We study the questions: Is this number finite? If so, what is the expected number of test case?

The program $\mathcal{P}$ gives rise to an infinite Markov chain, which is based on the SLD-tree corresponding to $\mathcal{P}$. Recall that $\mathcal{P}$ is governed by some probabilities `p_cont`, `p_1`, ..., which we will denote by p_c, $p_1, \ldots, p_r$. The Markov chain is depicted in Figure 1. Note that we model choice points via the states s_i. This is necessary, because $\mathcal{P}$ will backtrack when a call to `command1` fails, and proceed to `command2` with probability p_2. Double-circles denote *output states* – that is, whenever such a state is visited, a test case terminating in that state is generated. Node ♯ corresponds to the empty list. $\perp$ is the only absorbing state. It corresponds a termination of the resolution algorithm.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

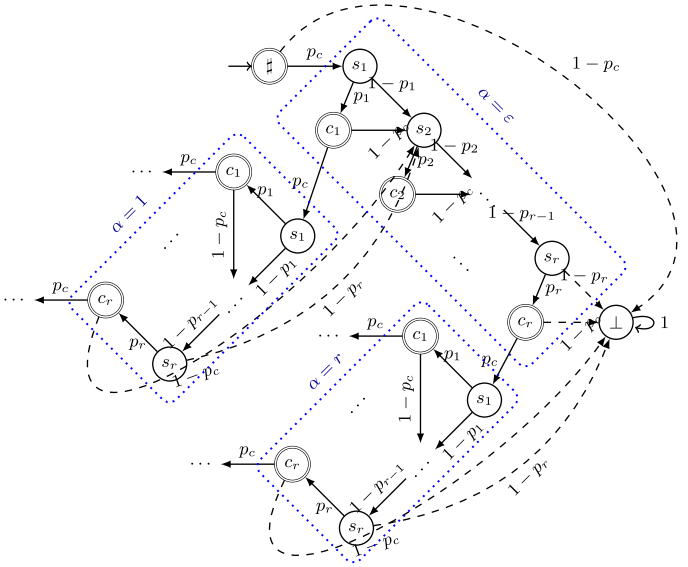


Fig. 1. The Markov chain corresponding to $\mathcal{P}$ with $\mathsf{Init} = \sharp$ and *blocks* $\alpha \in \{1, \ldots, r\}^*$ surrounded by blue boxes.

The blue boxes denote areas that share a common structure. We call these areas *blocks*. We can uniquely identify each block by a finite sequence $\alpha \in \{1, \ldots, r\}^*$. For any state s in the Markov chain, we denote by $\mathsf{Block}(x)$ the unique block that contains it. For any label occurring in a block ($s_1, s_2, \ldots, s_r$ and $c_1, \ldots, c_r$) and a block α, write s_1^α, c_1^α, ...for the unique state with that label in block α. In this way, we can identify any state in the Markov chain. Put differently, the Markov chain is given by a state space $\mathcal{S} = \{s_i^\alpha, c_i^\alpha \mid 1 \leq i \leq r,\ \alpha \in \{1, \ldots, r\}^*\} \cup \{\bot, \sharp\}$ and transition probabilities $p(s, s')$ for $s, s' \in \mathcal{S}$:

$$
\begin{aligned}
p(\sharp, s_1^\varepsilon) &= p_c && \\
p(\sharp, \bot) &= 1 - p_c && \\
p(s_i^\alpha, s_{i+1}^\alpha) &= 1 - p_i && 1 \leq i < r \\
p(c_i^\alpha, s_{i+1}^\alpha) &= 1 - p_c && 1 \leq i < r \\
p(c_i^\alpha, s_1^{\alpha \cdot i}) &= p_c && 1 \leq i \leq r \\
p(s_i^\alpha, c_i^\alpha) &= p_i && 1 \leq i \leq r
\end{aligned}
$$

The dashed *upward* arrows (which correspond to backtracking to a lower-recursion level) are somewhat more technical to define. Those arrows originate in states of the form s_r^α or c_r^α. There are several cases to consider:

a) $\alpha \in \{1, \ldots, r\}^* \cdot i$ for some $1 \leq i < r$
b) $\alpha \in \{1, \ldots, r\}^* \cdot i \cdot r^+$ for some $1 \leq i < r$
c) $\alpha \in r^*$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

This motivates the following transition probabilities

$$\begin{aligned}
p(s_r^{\alpha\cdot i}, s_{i+1}^{\alpha}) &= 1-p_r && 1\le i<r && \text{case a)}\\
p(c_r^{\alpha\cdot i}, s_{i+1}^{\alpha}) &= 1-p_c && 1\le i<r && \text{case a)}\\
p(s_r^{\alpha'\cdot i\cdot r\cdots r}, s_{i+1}^{\alpha'}) &= 1-p_r && 1\le i<r && \text{case b)}\\
p(c_r^{\alpha'\cdot i\cdot r\cdots r}, s_{i+1}^{\alpha'}) &= 1-p_c && 1\le i<r && \text{case b)}\\
p(s_r^{r\cdots r}, \bot) &= 1-p_r && && \text{case c)}\\
p(c_r^{r\cdots r}, \bot) &= 1-p_c && && \text{case c)}
\end{aligned}$$

We call edges from a block $\beta\cdot\alpha$ to a state in block β or from any block to $\bot$ an *upward* edge. They correspond precisely to the dashed arrows in Figure 1. If the Markov chain follows such an edge, we say block $\beta\cdot\alpha$ is *left upward* or that the chain *traverses upward* at that point. A block that has been left upward, is never visited again.

It is immediate that every state is visited at most once. There are no two states that can be reached from one-another. Note further that if we omit the dashed arrows and the state $\bot$, the resulting graph structure is an infinite, finitely branching tree. Yet, it is conceivable that the terminal state $\bot$ is never reached, because the sequence of states visited from $\sharp$ is infinite. The following proposition shows that this is not the case, provided $p_c<1$.

Proposition 1.
Let $s\in\mathcal{S}$ be any state. If $p_c<1$, then all sequences originating in s eventually leave $\mathsf{Block}(s)$ *upward. In particular, $\bot$ is visited eventually.*

Proof.
Let $\alpha=\mathsf{Block}(s)$. It is sufficient to show the result for $s=s_1^{\alpha}$. We first study the special case that there is an infinite path that *never* traverses upward. Pick an infinite path $s_0s_1s_2\cdots$ through the chain that never traverses upward. For every n, the prefix $s_0\cdots s_n$ must traverse at least $t_n\overset{\text{def}}{=}1+\lfloor\frac{n}{2r}\rfloor$ edges of the form $(c_i^{\beta}, s_1^{\beta\cdot i})$ (for correspondingly many distinct blocks β). This is because inside a block, there are only $2r$ states and no cycles. Hence, the probability of such a prefix is at most $p_c^{t_n}$ which tends to 0 as $n\to\infty$. As a result, the probability of any path that never traverses upward is 0.

Now for any i, consider the sub-tree of nodes below c_i^{α} that are visited. Since every node in the Markov chain can be visited at most once, the only option to remain in this tree indefinitely is for the tree to be infinite. However, the Markov chain is finitely branching. Therefore, the sub-tree of visited nodes below c_i^{α} is finitely branching. By König's lemma, this tree contains an infinite path and hence has probability 0. □

Corollary 1.
Let α be any block. The probability of reaching α from s_1^{ε} (i.e. from the initial block) is:

$$\Pr[H^{\alpha}<\infty]=p_c^{|\alpha|}\cdot\prod_{i=1}^{|\alpha|}p_{\alpha_i}<\infty$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Consequently, the probability of reaching α from $\sharp$ is $p_c^{|\alpha|+1} \cdot \prod_{i=1}^{|\alpha|} p_{\alpha_i}$.

Let $s \in \mathcal{S}$. We denote by $N(s)$ the random variable that counts the total number of states visited from s (including those in downstream blocks), before $\mathsf{Block}(s)$ is left upward. A useful observation is that $N(s) = H_s^E$ can also be expressed as a hitting time, where $E = \{s_i^\beta \mid \beta \prec \mathsf{Block}(s),\ 1 \le i \le r\} \cup \{\bot\}$. Note that it would suffice to take the subset of E which contains $s_{\alpha_i+1}^\beta$ for any $\beta = \alpha_1 \cdots \alpha_{i-1} \prec \alpha$. To define this set, we would have to work around the case $\alpha_i = r$ – indeed, if $\alpha \in r^*$, then $E = \{\bot\}$. So we define E as larger than needed purely to simplify notation. Note moreover that E depends on $\alpha = \mathsf{Block}(s)$. Since α is usually clear from context, we simply write E, but also use the notation E_α when needed.

Lemma 1.
Let $s \in \mathcal{S}$ and write $p_{\mathsf{max}} = \max\{p_1, \ldots, p_r\}$. If $p_c < 1$ and $\eta \stackrel{\text{def}}{=} r \cdot p_{\mathsf{max}} \cdot p_c < 1$, then $\mathsf{E}[N(s)]$ is finite.

Proof.
Let $\alpha = \mathsf{Block}(s)$. It is obvious that $N(x^\alpha) \le N(s_1^\alpha)$ for all $x \in \mathcal{S}$ with $\mathsf{Block}(x) = \alpha$. It therefore suffices to show that $N(s_1^\alpha)$ is finite. In the remainder of this proof, we write $\hat{s} = s_1^\alpha$.

Let now β be any block and let M_β denote the number of states visited in block β from s_1^β. Clearly $M_\beta \le 2r$. Let furthermore $I_\beta = \mathbb{1}_{H_{\hat{s}}^\beta < \infty}$ denote the indicator random-variable of the event that β is visited from $\hat{s}$. Note that both random-variables are independent because the underlying random events in $\mathcal{P}$ are independent and we count by M_β only states that are visited once β is entered. We have:

$$N(\hat{s}) = \sum_{\beta \in \alpha \cdot \{1,\ldots,r\}^*} I_\beta \cdot M_\beta$$

There are precisely r^l blocks that have distance $l \in \mathbb{N}$ from α. For each such block $\beta = \alpha \cdot \beta_1 \cdots \beta_l$, the probability of reaching it from α is $\Pr[I_\beta = 1] = p_c^l \cdot \prod_{i=1}^l p_{\beta_i}$ by Corollary 1 (if $l = 0$ then $\beta = \alpha$ and the probability is 1). Then $\Pr[I_\beta = 1] \le (p_{\mathsf{max}} \cdot p_c)^l$ for all β. This gives (using linearity of expectation and that I_β is independent from M_β for all β):

$$\begin{aligned}\mathsf{E}[N(\hat{s})] &= \sum_{\beta \in \alpha \cdot \{1,\ldots,r\}^*} \mathsf{E}[I_\beta] \cdot \mathsf{E}[M_\beta] \le \sum_{l=0}^{\infty} r^l \cdot (p_c^l \cdot p_{\mathsf{max}}^l) \cdot 2r \\ &= 2r \cdot \sum_{l=0}^{\infty} \eta^l = \frac{2r}{1-\eta}\end{aligned}$$

□

Let $s \in \mathcal{S}$. Denote by $O(s)$ the number of output states that are visited from s before $\mathsf{Block}(s)$ is left upward. Clearly $O(s) \le N(s)$.

Theorem 1.
Let $p_c < 1$ and $p_c \cdot r \cdot \max\{p_1, \ldots, p_r\} < 1$. Then for any block α

$$\mathsf{E}[O(s_1^\alpha)] = \frac{\sum p_i}{1 - p_c \sum p_i} \quad \text{and} \quad \mathsf{E}[N(s_1^\alpha)] = \frac{r + \sum p_i}{1 - p_c \sum p_i}$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Proof.
$C \stackrel{\text{def}}{=} \mathsf{E}[N(s_1^\alpha)]$ is finite by Lemma 1. Note that C is independent of α by the strong Markov property. We recall that $N(s_1^\alpha) = H^A_{s_1^\alpha}$ is a hitting time, where $A = \{s_i^\beta \mid \beta \prec \alpha\} \cup \{\bot\}$. In the remainder of the proof, we drop the superscript Greek letter for all states in α; that is, s_1 is understood to mean s_1^α.

Every path from s_1 to A must visit $s_2, \ldots, s_r$. Thus, by the strong Markov property, $N(s_1) = \left(\sum_{i=1}^{r-1} H^{s_{i+1}}_{s_i}\right) + H^A_{s_r}$. By linearity of expectation and Eq. (1)

$$\begin{aligned} C = \mathsf{E}[N(s_1)] &= \sum_{i=1}^{r-1} h^{s_{i+1}}_{s_i} + h^A_{s_r} = \sum_{i=1}^{r-1} 1 + p_i \cdot h^{s_{i+1}}_{c_i} + (1 + p_r \cdot h^A_{c_r}) \\ &= \sum_{i=1}^{r-1} 1 + p_i \left(1 + p_c \cdot \underbrace{h^{s_{i+1}}_{s_1^{\alpha \cdot i}}}_{C}\right) + \left(1 + p_r \left(1 + p_c \cdot \underbrace{h^A_{s_1^{\alpha \cdot r}}}_{C}\right)\right) \\ &= r + \sum_{i=1}^{r} p_i + C p_c \sum_{i=1}^{r} p_i \qquad (*) \end{aligned}$$

Solving for C proves the second claim of the theorem.

The proof for $\mathsf{E}[O(s_1)]$ is similar. We first make a slight modification to Eq. (1) to count only output states:

$$h^A_s = 0 \qquad \text{if } s \in A$$

$$h^A_s = \begin{cases} 1 + \sum_{s' \in \mathcal{S}} p(s, s') h^A_{s'} & s \text{ is an output state} \\ 0 + \sum_{s' \in \mathcal{S}} p(s, s') h^A_{s'} & s \text{ is not an output state} \end{cases} \qquad \text{if } s \notin A$$

This can be shown in exactly the same way as equations (1) (see e.g. Norris (1998) for the proof of the classical theorem; the adaption is straightforward). Alternatively, the following intuition can be turned into a formal proof:

Observe that in counting only output states $\mathcal{O} \subseteq \mathcal{S}$, we are effectively studying a second Markov chain, whose state set consists only of output states (and $\bot$). For $s, s' \in \mathcal{S}$ write $\mathsf{P}(s, s')$ for the set of all *simple paths* (without repeating vertices) from s to s' that do not visit $\mathcal{O}$. The transition probabilities p' of this second chain are given by the relation

$$p'(s, s') = \sum_{w \in \mathsf{P}(s, s')} \prod_{i=1}^{|w|-1} p(w_{i-1}, w_i)$$

So our modified formulas are simply a different way to write Eq. (1) for this modified chain in an iterative fashion.

With these modifications, we see that $(*)$ becomes:

$$C' = \sum_{i=1}^{r} p_i + C' p_c \sum_{i=1}^{r} p_i$$

Again, solving for C' establishes the claim. □

Note that for any given state s_1^α, the mean hitting time $h^{s_1^\alpha}_\sharp = \sum_{n \geq 1} \Pr[H^{s_1^\alpha}_\sharp \geq n] \geq \sum_{n \geq 1} 1 - p_c = \infty$ (where we use $\mathsf{E}[X] = \sum_{n \geq 1} \Pr[X \geq n]$ for any random variable that

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

only takes on positive integer values). So although we have a non-zero probability of selecting every test, we won't, informally speaking, do so on average. Naturally, this is solved by repeating the experiment a sufficient number of times. This is the content of the next section.

3.1.2 Infinite looping and time-to-hit

As shown in Lemma 1, the program in listing 1 terminates eventually. As a result, every state except $\perp$ in the Markov chain we studied above is transient and, moreover, the number of produced test cases is always finite. In testing, one aims at a high test coverage, and the number of test cases we produce in this fashion, though free of duplicates, has a low chance of visiting tests in deep blocks. A natural approach is to loop on the predicate `t/1` like so:

```
main_loop(X) :- repeat, t(X).
```

With respect to our Markov chain this amounts to removing $\perp$ and to instead redirect any arc into $\perp$ to $\sharp$. The resulting chain is recurrent (indeed positive recurrent) and we compute the mean hitting time of any state. In what follows, we will assume $p_1 = p_2 = \cdots = p_r \stackrel{\text{def}}{=} p$ such that $r \cdot p \cdot p_c < 1$ (as in Theorem 1). Moreover, we assume that $p(\sharp, s_1^{\varepsilon}) = 1$, so that the empty list is never selected as an output. This simplifies the formulas below slightly, but has otherwise no effect on the line of reasoning we give here.

Given the conditions of Theorem 1, there is a constant $C = N(s_1^{\alpha})$ that is independent of the value of α. As noted before, $C = h_{s_1^{\alpha}}^{E_{\alpha}}$ is a mean hitting time where $E_{\alpha} = \{s_i^{\beta} \mid \beta \prec \alpha,\ 1 \le i \le r\} \cup \{\sharp\}$ (note that we modified the definition of E used in the previous section by replacing $\perp$ by $\sharp$). Recall that we usually drop the subscript α, because it is clear from context.

If we hop from one state s_i^{α} to its neighbor s_{i+1}^{α} we might traverse the tree below $s_1^{\alpha \cdot i}$ with probability $p \cdot p_c$. That step will visit C states. This means (by Eq. (1)):

$$h_{s_i^{\alpha}}^{s_{i+1}^{\alpha}} = 1 + p(1 + p_c C) \stackrel{\text{def}}{=} \Delta$$

More generally the mean hitting time within a block is again independent of α and can be computed as:

$$h_{s_1^{\alpha}}^{s_i^{\alpha}} = h_{s_1^{\alpha}}^{s_2^{\alpha}} + h_{s_2^{\alpha}}^{s_3^{\alpha}} + \cdots + h_{s_{i-1}^{\alpha}}^{s_i^{\alpha}} = (i-1)\Delta \tag{2}$$

We define the *leave upward time* $U_{s_i^{\alpha}} = h_{s_i^{\alpha}}^{E}$ where $E = E_{\alpha}$ as above. Note that the value $U_{s_i^{\alpha}} \in \mathbb{N} \cup \{\infty\}$ does not actually depend on α. This justifies writing $U_i = U_{s_i^{\alpha}}$. It is obvious that $C = U_1$. Moreover, by using the same derivation as that in Eq. (2):

$$U_i = (r - i + 1)\Delta \quad 1 \le i \le r \tag{3}$$

We already noted that $C = U_1$. A related quantity is the hitting time of $\sharp$ from any s_i^{α}, $\alpha = \alpha_1 \cdots \alpha_t$, which we may compute using the intermediate leave upward times:

$$h_{s_i^{\alpha}}^{\sharp} = U_i + U_{\alpha_t + 1} + \cdots U_{\alpha_1 + 1}$$

Note that we abuse notation: Equation (3) gives $U_{r+1} = 0$. While s_{r+1}^{α} does not exist and hence the corresponding hitting time is not defined, it is convenient to allow such terms and exploit that $U_{\alpha_j + 1} = 0$ whenever $\alpha_j = r$ $(1 \le j \le t)$.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

With this, we may compute:

$$h^{\sharp}_{s^{\alpha}_i} = U_i + \sum_{k=1}^{t} U_{\alpha_k+1} = \Delta \cdot \left((r-i+1) + \sum_{k=1}^{t} (r-(\alpha_k+1)+1) \right)$$
$$= \Delta \cdot \left(1 + \sum_{k=0}^{t} r - \alpha_k \right) \qquad \text{where } \alpha_0 \stackrel{\text{def}}{=} i \tag{4}$$

Note again that the formula works correctly, if $i = r+1$: Say $\alpha = rrr$. Then we are in the process of falling back to $\sharp$ and the equation gives 0. While the hitting time is again not defined for the non-existent state s_{r+1}, we will sometimes have to compute the hitting time of $\sharp$ from the "right neighbor" of s_{i+1}. In these situations, abusing notation in this way is useful because we need not distinguish between cases where $i < r$ and those where $i = r$.

Finally, we may now compute the hitting time of an arbitrary state in terms of hitting times in intermediate blocks, again using Eq. (1). Let $\alpha = \beta \cdot j$.

$$\begin{aligned} h^{s^{\alpha}_1}_{\sharp} &= h^{s^{\beta}_j}_{\sharp} + 1 \\ &\quad + (1-p)\left(h^{\sharp}_{s^{\beta}_{j+1}} + h^{s^{\alpha}_i}_{\sharp} \right) && \text{(fall through to } s^{\beta}_{j+1}\text{)} \\ &\quad + p\left(1 + (1-p_c)\left(h^{\sharp}_{s^{\beta}_{j+1}} + h^{s^{\alpha}_i}_{\sharp} \right) \right) && \text{(no visit to next block } \alpha\text{)} \\ &= h^{s^{\beta}_j}_{\sharp} + 1 + p + \left(h^{\sharp}_{s^{\beta}_{j+1}} + h^{s^{\alpha}_i}_{\sharp} \right)(1-pp_c) \end{aligned}$$

This gives

$$h^{s^{\alpha}_1}_{\sharp} = \frac{h^{s^{\beta}_j}_{\sharp} + 1 + p + (1-pp_c) h^{\sharp}_{s^{\beta}_{j+1}}}{pp_c}$$

and together with Eq. (2) and Eq. (4), recalling that $\alpha_{|\alpha|} = j$, we have:

$$\begin{aligned} h^{s^{\alpha}_i}_{\sharp} &= \frac{h^{s^{\beta}_j}_{\sharp} + 1 + p + (1-pp_c) h^{\sharp}_{s^{\beta}_{j+1}}}{pp_c} + (i-1)\Delta \\ &= \frac{h^{s^{\beta}_j}_{\sharp}}{pp_c} + \frac{1}{pp_c}\left(1 + p + (1-pp_c)\left(\sum_{k=1}^{|\alpha|} r - \alpha_k \right)\Delta \right) + (i-1)\Delta \end{aligned} \tag{5}$$

The following theorem gives a closed formula:

Theorem 2.
Let s^{α}_i for some $\alpha = \alpha_1 \cdots \alpha_t$. Let $\nu = pp_c$ and $\nu \cdot r < 1$. Then

$$h^{s^{\alpha}_i}_{\sharp} = \nu^{-t} + (i-1)\Delta + \sum_{k=1}^{t} \frac{1 + p + (\alpha_k - 1)\Delta + (1-\nu)\sum_{s=1}^{k}(r-\alpha_s)\Delta}{\nu^{t+1-k}} \tag{6}$$

Proof.
By induction on t. If $t = 0$, then $\alpha = \varepsilon$ and by Eq. (2), we have $h^{s^{\varepsilon}_i}_{\sharp} = 1 + (i-1)\Delta$. Moreover the empty sum in Eq. (6) equates to 0 establishing the induction base.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Now let $t > 0$ and assume the statement holds for $t-1$. By induction, we may replace $h_\sharp^{s_j^\beta}$ in Eq. (5) with Eq. (6):

$$\begin{aligned}
&\frac{1}{\nu}\left(\nu^{-(t-1)} + (\alpha_t - 1)\Delta + \sum_{k=1}^{t-1} \frac{1 + p + (\alpha_k - 1)\Delta + (1-\nu)\sum_{s=1}^{k}(r-\alpha_s)\Delta)}{\nu^{t-k}}\right) \\
&\quad + \frac{1}{\nu}\left(1 + p + (1-\nu)\left(\sum_{s=1}^{t} r - \alpha_s\right)\Delta\right) + (i-1)\Delta \\
&= \nu^{-t} + \sum_{k=1}^{t-1} \frac{1 + p + (\alpha_k - 1)\Delta + (1-\nu)\sum_{s=1}^{k}(r-\alpha_s)\Delta)}{\nu^{t+1-k}} \\
&\quad + \frac{(1 + p + (\alpha_t - 1)\Delta + (1-\nu)(\sum_{s=1}^{t} r - \alpha_s)\Delta)}{\nu^{t+1-t}} + (i-1)\Delta \\
&= \nu^{-t} + (i-1)\Delta + \sum_{k=1}^{t} \frac{1 + p + (\alpha_k - 1)\Delta + (1-\nu)\sum_{s=1}^{k}(r-\alpha_s)\Delta}{\nu^{t+1-k}}
\end{aligned}$$

□

Corollary 2.
Let $\nu = p \cdot p_c$ with $\nu \cdot r < 1$. Then $h_\sharp^{s_i^\alpha} \in \Theta(\nu^{-t})$ for any $\alpha = \alpha_1 \cdots \alpha_t$.

Proof.
Write Eq. (6) as

$$\nu^{-t} + A + \nu^{-t-1} \sum_{k=1}^{t} \frac{B_k + \sum_{s=1}^{k} D_s}{\nu^{-k}}$$

for suitable constants (in t) $A \geq 0$, $B_k \geq 0$, and $D_s \geq 0$, whereby $h_\sharp^{s_i^\alpha} \in \Omega(\nu^{-t})$.

Choose suitable largest values $B \geq B_k$ for all $t \in \mathbb{N}$, $1 \leq k \leq t$, and $D \geq D_s$ for all $1 \leq s \leq t$. Bound Eq. (6) from above by

$$\nu^{-t} + A + \nu^{-t-1} \sum_{k=1}^{t} \frac{B + D \cdot k}{\nu^{-k}} \leq \nu^{-t} + A + \nu^{-t-1}(B+D) \cdot \sum_{k=1}^{t} \frac{k}{\nu^{-k}}$$

A well-known calculation via derivatives gives $\sum_{k=1}^{t} k \cdot \nu^k = \nu \sum_{k=1}^{t} k\nu^{k-1} \leq \nu \sum_{k=1}^{\infty} k\nu^{k-1} = \nu \cdot \frac{\mathrm{d}}{\mathrm{d}\nu} \sum_{k=0}^{\infty} \nu^k = \frac{\nu}{(1-\nu)^2}$. With that we have

$$\nu^{-t} + A + \nu^{-t-1} \sum_{k=1}^{t} \frac{B + D \cdot k}{\nu^{-k}} \leq \nu^{-t} + A + (B+D)\frac{\nu^{-t}}{(1-\nu)^2} \in \mathcal{O}(\nu^{-t})$$

□

3.2 Drop-and-Shuffle strategy

3.2.1 Number of generated tests

To study the number of generated tests in this context, we again have to define a Markov chain. The Markov chain in the shuffle and drop scenario is significantly more complicated than the one we studied previously in subsection 3.1.1. This is because there are now multiple ways a given test case can be output. To distinguish between those, we need to use a larger and more complex state set. We will illustrate this, before defining the Markov chain formally.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Consider again the program in listing 1. When the current goal is `command(H)`, we have a set R of rules whose head unifies with this goal. In this case, $R = \{$`command(H) :- command1(H)`$, \ldots,$ `command(H) :- commandr(H)`$\}$. Recall this is meant to represent $r \in \mathbb{N}$ distinct rules. In standard SLD-resolution, we would select the first rule that occurs in the input program $\mathcal{P}$, namely `command(H) :- command1(H)` first, and push the remaining $r-1$ rules on the stack from right to left. During backtracking, we would then eventually explore each of those rules in the order given in the input program (except in case of an infinite recursion).

In the drop-and-shuffle strategy, we first perform an independent Bernoulli trial for each rule $\rho \in R$: With probability p_d, we remove ρ from R. We refer to p_d as the *drop probability*. In this way, a set $R' \subseteq R$ is computed. The random variable $|R'|$ follows a Binomial distribution: $\Pr[|R'| = k] = \binom{|R|}{k} p_d^{|R|-k}(1-p_d)^k$. Next, we shuffle the set R'. To this end, we select a permutation $\pi \in \mathbb{S}(R')$, where $\mathbb{S}(M)$ denotes the symmetric group on a given set M. Conceptually, any probability distribution on $\mathbb{S}(R')$ is conceivable. In this paper, we follow a simpler approach and select π uniformly at random from $\mathbb{S}(R')$. In this way, we obtain an ordered tuple of elements of R without any repetitions.

The result of these two random processes is a tuple $(\rho_{i_1}, \ldots, \rho_{i_k})$ where $k = |R'|$ and $\rho_{i_j} \in R$. To simplify notation, we identify R' with this tuple in what follows. Since both random events – dropping and shuffling – are independent, the probability of each such tuple $R' = (\rho_{i_1}, \ldots, \rho_{i_k})$ is precisely $\Pr[R'] = \frac{p_d^{n-k}(1-p_d)^k}{k!}$.

Remark 1.
Note that this random process is different from the classical "drawing without replacement", where the number of elements that are drawn is usually a fixed parameter.

These observations motivate the following Markov chain $\mathcal{M} = (\mathcal{S}, (X_n)_{n\in\mathbb{N}_0}, p, \varepsilon)$. The state set $\mathcal{S}$ now consists of stacks of choice-points – in loosely the same way a Prolog runtime would maintain them. Before formally defining the state set and transition probabilities, we invite the reader to consider a simplified graphical representation of the chain, as given in Figure 2.

Figure 2 gives an overview of the chain. Some details have been omitted or simplified to avoid cluttering the picture. Probabilities are not shown. We have omitted choice points from a higher layer: A state/stack of the form $[H|T][](1,5,2,3)[H|T](1,2,3)$ is thus simply represented as $(1,2,3)$, omitting the "lower" parts of the stack. Note that these items are implicitly clear from the path to a given node. Moreover, most backtracking arrows have been omitted. Finally, at any given depth, both the node $[H|T][]$ and the node $[H|T]$ each have $\sum_{k=0}^{r} \binom{r}{k} \cdot k! = \lfloor r! \cdot \mathsf{e} \rfloor$ children.[2] These have also mostly been omitted.

We describe the state set via (the language defined by) a regular expression. First we need two auxiliary languages:

$$\mathcal{S}_{\mathsf{Sel}} = \{[], [H|T], [][H|T], [H|T][]\}$$
$$\mathcal{S}_{\mathsf{Com}} = \{(y_1, \ldots, y_l) \mid 1 \le l \le r,\ y_i \in \{1, \ldots, r\}, y_i \neq y_j \text{ for all } i \neq j\}$$

The set $\mathcal{S}_{\mathsf{Com}}$ corresponds to all ordered subsets of `command` rules, that is, subsets of $\{1, \ldots, r\}$. Note that the empty subset is excluded, that is, $() \notin \mathcal{S}_{\mathsf{Com}}$. The set $\mathcal{S}_{\mathsf{Sel}}$

[2] We have $\sum_{k=0}^{r} \binom{r}{k} k! = r! \sum_{k=0}^{r} \frac{1}{k!} \approx r! \cdot \mathsf{e}$ with error term $\sum_{k=r+1}^{\infty} \frac{1}{k!} \in (0,1)$, because $r \ge 1$.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

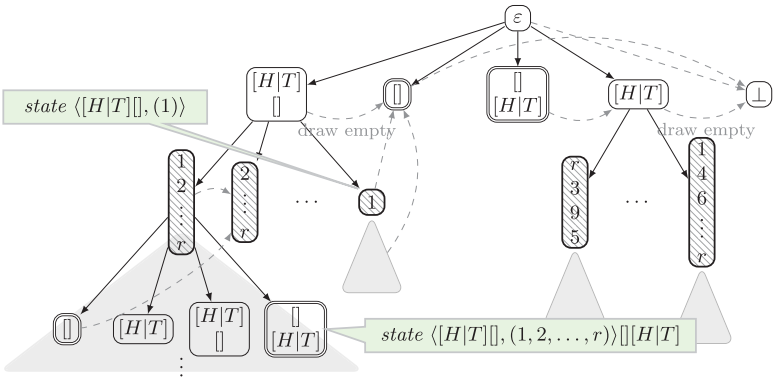


Fig. 2. The Markov chain corresponding to $\mathcal{P}$ in the Drop-and-Shuffle approach. Dashed arrows represent backtracking. Double circled nodes produce an output. Hatched nodes are *recursive sub-tree roots*, which start an entire infinite sub-tree with the same structure as the whole chain (shown as gray triangles).

corresponds to the five probabilistic options the drop-and-shuffle algorithm gives us for resolving goals of the form `t(X)`, including potential choice-points for backtracking. Note that, again, the "empty selection" $() \notin \mathcal{S}_{\mathsf{Sel}}$ is not included.

We can now define the state set $\mathcal{S} = (\mathcal{S}_{\mathsf{Sel}} \times \mathcal{S}_{\mathsf{Com}})^* \cdot (\mathcal{S}_{\mathsf{Sel}} + \varepsilon)$. The "empty state" $\varepsilon \in \mathcal{S}$ is the initial state of the chain; that is, $\mathsf{Init} = \varepsilon$. We write the pairs $\langle x, y\rangle \in \mathcal{S}_{\mathsf{Sel}} \times \mathcal{S}_{\mathsf{Com}}$ in angular brackets in order to visually distinguish them and improve readability.

Given a state $\varepsilon \neq s \in \mathcal{S}$, we may write it as $s = w \cdot x$ or $s = w \cdot \langle x, y\rangle$ with $w \in \mathcal{S}$, $x \in \mathcal{S}_{\mathsf{Sel}}$, and $y \in \mathcal{S}_{\mathsf{Com}}$. Because $\varepsilon \notin \mathcal{S}_{\mathsf{Sel}}$ and also $\varepsilon \notin \mathcal{S}_{\mathsf{Com}}$, this factorization is unique. We make liberal use of this observation when defining the transition probabilities. We first define transitions that *descend* further into the tree. These correspond to solid arrows in Figure 2. For any $l \geq 1$, $w \in \mathcal{S}$, and $x \in \mathcal{S}_{\mathsf{Sel}}$:

$$p(w,\ w \cdot x) = \frac{(1-p_d)^2}{2} \qquad x \in \{[H|T][], [][H|T]\}$$

$$p(w,\ w \cdot x) = (1-p_d)p_d \qquad x \in \{[], [H|T]\}$$

$$p(w \cdot x,\ w \cdot \langle x, (y_1, \ldots, y_l)\rangle) = \frac{p_d^{r-l}(1-p_d)^l}{l!} \qquad x \in \{[H|T][], [H|T]\}$$

Next, we define transitions that correspond to *backtracking*. These correspond to dashed arrows in Figure 2. To this end we define the operation $\mathsf{Pop}\colon \mathcal{S} \to \mathcal{S}$. Intuitively, this operation pops from the stack until we arrive at a previously unpursued choice point. Looking back at Figure 2, it identifies the target of the backtracking arrow. It may be necessary to remove multiple layers of pairs $\langle x, y\rangle$ when backtracking. For example $\mathsf{Pop}(\langle [H|T][], (1,3)\rangle\langle [H|T][], (3)\rangle\langle [H|T], (1)\rangle) = \langle [H|T][], (1,3)\rangle[]$. Formally, we define Pop recursively with $\mathsf{Pop}(\varepsilon) = \bot$ and:

$$\mathsf{Pop}(w \cdot [][H|T]) = w \cdot [H|T] \qquad \mathsf{Pop}(w \cdot [H|T][]) = w \cdot []$$

$$\mathsf{Pop}(w \cdot []) = \mathsf{Pop}(w) \qquad \mathsf{Pop}(w \cdot [H|T]) = \mathsf{Pop}(w)$$

$$\mathsf{Pop}(w \cdot \langle x, (y_1)\rangle) = \mathsf{Pop}(w \cdot x) \qquad \mathsf{Pop}(w \cdot \langle x, (y_1, y_2, \ldots, y_l)\rangle) = w \cdot \langle x, (y_2, \ldots, y_l)\rangle$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

We can now define all backtracking transitions as follows:

$$p(w \cdot x, \mathsf{Pop}(w \cdot x)) = 1 \qquad x = [] \text{ or } x = [][H|T]$$
$$p(w \cdot x, \mathsf{Pop}(w \cdot x)) = p_d^r \qquad x = [H|T] \text{ or } x = [H|T][]$$
$$p(w, \mathsf{Pop}(w)) = p_d^2 \qquad w = w' \cdot \langle x, y\rangle \text{ or } w = \varepsilon$$

The last two model the event that all matching rules are dropped when unifying `command(X)` or `t(X)`, respectively.

Note the recursive structure of $\mathcal{M}$: Any state of the form $w \cdot \langle x, y\rangle$ ($x \in \mathcal{S}_{\mathsf{Sel}}$, $y \in \mathcal{S}_{\mathsf{Com}}$) is the root of an infinite sub-tree that has a structure identical to that of $\mathcal{M}$. We call such states *recursive sub-tree roots* or simply *sub-tree roots*. These states are drawn with hatched background in Figure 2.

To each recursive sub-tree root s corresponds a unique state Exit_s that is visited when the chain leaves that sub-tree (via backtracking). In other words: All paths that exit the sub-tree below s must traverse Exit_s. For example, consider the left-most gray sub-tree in Figure 1 (with the large gray triangle in the background) below $s = \langle [H|T][], (1, 2, \ldots, r)\rangle$. Its exit-state is the next node to the right: $\mathsf{Exit}_s = \langle [H|T][], (2, \ldots, r)\rangle$. In general, if s is of the form $w\langle x, y\rangle$, then $\mathsf{Exit}_s = \mathsf{Pop}(w \cdot \langle x, y\rangle)$. Note that Exit_s may be at the same depth as s or at a lower depth than s. It is never at a higher depth.

We can now compute the average number of generated tests as before. It is again the mean hitting time $h_\varepsilon^\perp$. By an argument identical to Proposition 1, the chain will reach $\perp$ eventually with probability 1, if $p_d > 0$. But that does *not* imply that the hitting time – an expected value – converges for all such values of p_d. Indeed, the hitting time is finite only for a subset of possible choices for $p_d > 0$, as the following theorem shows:

Theorem 3.
Let $p_d \in (1 - \frac{1}{\sqrt{r}}, 1]$. Then the expected number $h_\varepsilon^\perp$ of states visited from ε is finite and given by:

$$h_\varepsilon^\perp = \frac{1 + 2(1 - p_d)}{1 - r(1 - p_d)^2}$$

Moreover this hitting time is identical to $h_s^{\mathsf{Exit}_s}$ for any recursive sub-tree root s and its corresponding exit state Exit_s. We define $C \stackrel{\text{def}}{=} h_\varepsilon^\perp$ and remark that it is a constant property of the chain.

Proof.
The fact that $h_\varepsilon^\perp = h_s^{\mathsf{Exit}_s}$ for any recursive sub-tree root s is apparent from the definition of the chain: The transition probabilities are prefix invariant. So $p(x \cdot a, x \cdot b) = p(a, b)$ for all factorizations $s = xa$ with $x \in \mathcal{S}$. $h_\varepsilon^\perp$ is the unique minimal positive solution to the equations Eq. (1). Now, letting $C \stackrel{\text{def}}{=} h_\varepsilon^\perp$:

$$C = 1 + p_d(1 - p_d) \cdot 1 + p_d^2 \cdot 0$$
$$+ p_d(1 - p_d) \cdot \left(1 + \sum_{k=0}^{r} \frac{p_d^{r-k}(1 - p_d)^k}{k!} \cdot k! \cdot \binom{r}{k} \cdot k \cdot C\right)$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

$$+\frac{1}{2}\cdot(1-p_d)^2\cdot\left(2+\sum_{k=0}^{r}\frac{p_d^{r-k}(1-p_d)^k}{k!}\cdot k!\cdot\binom{r}{k}\cdot k\cdot C\right)$$

$$+\frac{1}{2}\cdot(1-p_d)^2\cdot\left(1+\sum_{k=0}^{r}\frac{p_d^{r-k}(1-p_d)^k}{k!}\cdot k!\cdot\binom{r}{k}\cdot(k\cdot C+1)\right)$$

We recall that $(x+y)^r=\sum_{k=0}^{r}\binom{r}{k}x^k y^{r-k}$. Differentiation and subsequent multiplication by x gives $rx(x+y)^{r-1}=\sum_{k=0}^{r}k\binom{r}{k}x^k y^{r-k}$. Moreover, recall that $\sum_{k=0}^{r}\binom{r}{k}p_d^{r-k}(1-p_d)^k=1$. With this, the above simplifies to

$$C=1+2(1-p_d)+Cr(1-p_d)^2$$

Now if $p_d\le 1-\frac{1}{\sqrt{r}}$, then this implies $C\ge 1+\frac{2}{\sqrt{r}}+C$, which is possible only if $C=\infty$. On the other hand, if $p_d\in(1-\frac{1}{\sqrt{r}},1]$ then solving for C establishes the claim. Note that on this interval, the formula has no singularities and is positive. □

3.2.2 Infinite looping and time-to-hit

We again construct a recursive analysis of the mean-hitting-time. Recall that in subsection 3.1.2 we analyzed the hitting time of the unique state corresponding to $\tau=(\tau_1,\ldots,\tau_l)$ (with $\tau_i\in\{1,\ldots,r\}$) by first considering the hitting time of the unique state corresponding to $\tau^{(l-1)}=(\tau_1,\ldots,\tau_{l-1})$, and then constructing the full hitting time from that number "bottom-up". This recursive argument works less well in the present scenario, where many different states correspond to $\tau^{(l-1)}$. Computing the overall hitting time as a sum of those intermediate hitting times is challenging, as we explain below. We therefore develop an approach to compute the hitting time "top-down".

First, we need to add looping to the Markov chain, as in subsection 3.1.2. Recall that there we merged the two states ♯ and ⊥. We do *not* do that here. Instead, we add an edge from ⊥ to ε with probability 1. This is for technical reasons, and we will justify this choice further below.

When running the program $\mathcal{P}$ in the drop-and-shuffle strategy, first we randomly select and permute a subset of the two rule-heads `t([])` and `t([H–T])`. To proceed, we need to visit $[H|T]$ or $[H|T][]$ at depth 0. Once that happens, there are two options: With probability p_d, the next state does *not* contain τ_1, and thus we *cannot* visit τ in this loop iteration (i.e. before returning to ε first). We call such a sub-tree root *unproductive*. Conversely, with probability $(1-p_d)$, the next state does contain τ_1 somewhere within its stack (though not necessarily at the top). Those sub-tree roots and their corresponding sub-trees are called *productive* (shown in green in Figure 3). If a productive or unproductive state does not have τ_1 at its top, there is an infinite tree below it that will we be explored, but cannot yield the desired test case. We also call such a sub-tree *unproductive* (shown in gray in Figure 3). Note that productive sets of size $k>0$ give rise to precisely $k-1$ unproductive sub-trees, though not all of those are traversed *before* the sub-tree containing τ is visited. Let A_τ denote the set of states that output τ:

Fact 1.
If the chain arrives at an unproductive state, it must first visit ε before visiting A_τ.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

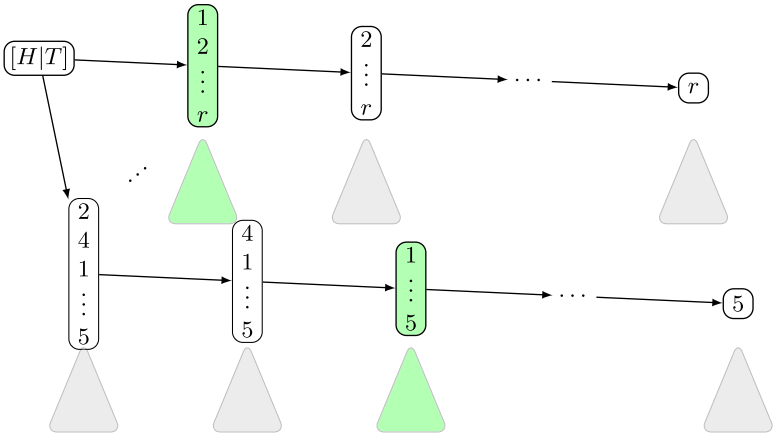


Fig. 3. Recursive sub-tree roots are connected to *all* ordered subsets of $\{1, \dots, r\}$ (most arrows omitted for readability). Each subset of size k starts *chain* of k subsets. One element (here 1) is the desired next item of the test sequence, and each chain contains at most one state with the desired item at the top (depicted in green).

Note that we are only talking about depth 0 for now, so this fact is obvious. At higher depths, we would need to adjust the definition of "productive" to ensure that all prefixes at lower depths have $\tau_1, \tau_2, \dots$ at the top.

If the chain arrives at a productive state, the situation is more complex: The chain needs to traverse a number of unproductive sub-trees, depending on the position of τ_1 in the ordered set. For example, in the second branch shown in Figure 3, there are two unproductive sub-trees to be traversed *before* reaching productive sub-tree (and several, indicated by the dots, after). Below the productive state, we find a tree of recursive structure: The goal now is to reach a test case of length $l-1$, or to return to ε and start from scratch (cf. Fact 1).

Starting from a productive state s of size $m \leq r$ with item τ_1 at position $k \in \{0, \dots, m\}$, the hitting time is thus $kC + h_x^{A_\tau}$, where C is the quantity from Theorem 3 and $x \in \mathcal{S}$ is the state arising from s after k items have been popped (e.g. the green state in the lower branch in Figure 3).

We stress that $h_x^{A_\tau}$ depends not only on the remaining test-case suffix $(\tau_2, \dots, \tau_l)$, but also on the number of (unproductive) backtracking steps – at least $(m-k-1)C$ – before returning to ε. The chain may arrive at an unproductive step at some point, and by Fact 1, it first needs to visit ε before reaching A_τ. Note that this may happen at depth >1 as well! This prevents us from applying induction in a straightforward way. We thus need the following lemma, which allows us to reduce the hitting time $h_x^{A_\tau}$ to three quantities we may compute individually and inductively:

Lemma 2.
For any recurrent Markov chain, let $A \subseteq \mathcal{S}$ and $x, y \in \mathcal{S} \setminus A$ with $x \neq y$. Then:

$$h_x^A = h_x^{A \cup \{y\}} + \Pr[H_x^y < H_x^A] \cdot h_y^A$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

We prove the lemma at the end of this section, as it is purely Markov theoretic. In our immediate setting, the consequence is that

$$h_\varepsilon^{A_\tau} = h_\varepsilon^{A_\tau \cup \{\bot\}} + q \cdot h_\bot^{A_\tau} = h_\varepsilon^{A_\tau \cup \{\bot\}} + q(1 + h_\varepsilon^{A_\tau})$$
$$\iff h_\varepsilon^{A_\tau} = \frac{h_\varepsilon^{A_\tau \cup \{\bot\}} + q}{1 - q} \tag{7}$$

for some probability q that depends on τ. We shall see below that q in fact only depends on the length l of the test case. Note that by not merging $\bot$ and ε, we fulfill the "$x \neq y$ premise" of the lemma. By choosing $p_d \in (1 - \frac{1}{\sqrt{r}}, 1)$, we obtain a recurrent chain.

It is thus sufficient to compute q and $h_\varepsilon^{A_\tau \cup \{\bot\}}$. We have "widened" the set A_τ by including $\bot$, which means we no longer have to worry about looping back to ε. This effectively allows us to compute the desired hitting-time inductively, applying Lemma 2 iteratively. We first turn to computing q as a function of τ.

Let $s = \langle x_1, y_1 \rangle \cdots \langle x_m, y_m \rangle \in \mathcal{S}$ be any recursive sub-tree root, where $y_i = (y_{i,1}, \ldots, y_{i,n_i}) \in \mathcal{S}_{\mathsf{Com}}$ for $1 \leq i \leq m$. Then the sequence $y_{1,1} \cdots y_{m,1}$ is the *prefix* of s. It corresponds to the sequence that would be output if state $s \cdot []$ is reached.

Proposition 2.

1. *Let s be a recursive sub-tree root with prefix ρ and $\tau = (\tau_1, \ldots, \tau_l)$. Then*
$$\Pr[H_s^{A_{\rho\tau}} > H_s^{\mathsf{Exit}_s}] = 1 - (1 - p_d)^{2l+1} \stackrel{\text{def}}{=} q^{(l)}$$
In particular, this probability depends only on l and not on the values τ_i or ρ.
2. *Let s be a recursive sub-tree root with prefix ρ and let τ be any test case. Then:*
$$h_s^{A_{\rho\tau} \cup \{\mathsf{Exit}_s\}} = h_\varepsilon^{A_\tau \cup \{\bot\}}$$
Informally, the stated mean hitting time is "translation invariant".

Proof.

1. If $l = 0$, then $\Pr[H_s^{A_\rho} > H_s^{\mathsf{Exit}_s}] = (1 - p_d)p_d + p_d^2 = p_d$, which is the probability of not selecting [], $[H|T][]$ or $[][H|T]$, each of which would eventually output the prefix ρ. This depends only on $l = 0$, so write $q^{(0)} = p_d$.
Now let $l > 0$: We reach Exit_s without visiting the next deeper state by backtracking to it immediately with probability p_d^2, or by visiting [] with probability $(1 - p_d)p_d$ which sums up to p_d. Otherwise we reach the next deeper state with probability $(1 - p_d)$. Here we either select τ_1 and apply induction using Fact 1, or we do not select τ_1. This gives:
$$\Pr[H_s^{A_{\rho\tau}} > H_s^{\mathsf{Exit}_s}] = p_d + (1 - p_d)((1 - p_d) \cdot q^{(l-1)} + p_d)$$
which is of the form $A \cdot q^{(l-1)} + B$ with $A = (1 - p_d)^2$ and $B = p_d(2 - p_d)$. This gives rise to a polynomial (in A), namely: $q^{(0)} \cdot A^l + B \cdot \sum_{k=0}^{l-1} A^k$. Computing the geometric sum and simplifying the term proves the first claim.
2. This follows immediately from the definition of Exit_s and the recursive structure of the chain.

□

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

In what follows, we write $\tau^{(i)} = (\tau_1, \dots, \tau_i)$. So $\tau = \tau_l$ and $\tau_0 = ()$ is the empty test sequence. Write A_i for the set of states that output $\tau^{(i)}$. For simplicity, we write $h_i = h_\varepsilon^{A_i \cup \{\bot\}}$.

Lemma 3.
Let $p_d \in (1 - \frac{1}{\sqrt{r}}, 1)$ and $i \in \{0, \dots, l\}$ and let C denote the constant from Theorem 3. Then

$$h_i = (1-p_d)^{2i} \left(1 + \frac{(1-p_d^2)(1+C)}{2} - \frac{C(1-p_d)^2(r-1)}{p_d(2-p_d)} - \frac{(1-p)(1+p)+(1-p)^3}{2p_d(2-p_d)} - i \cdot (1-p_d)^2 \cdot \frac{1+C(r-1)}{2}\right) + \left(\frac{C(1-p_d)^2(r-1)}{p_d(2-p_d)} + \frac{(1-p)(1+p)+(1-p)^3}{2p_d(2-p_d)}\right)$$

Proof.
For h_0 we need the hitting time of the set of four states $\bot$, $[]$, $[H|T][]$, and $[][H|T]$. By Eq. (1) we have

$$h_0 = 1 + (1-p_d)p_d(1+C) + \frac{1}{2}(1-p_d)^2(1+C) = 1 + \frac{1}{2}(1-p_d^2)(1+C)$$

To compute h_{i+1}, we first look at the hitting time starting from the states $[H|T]$, $[H|T][]$, and $[][H|T]$. We select a productive state with probability $(1-p_d)$ and an unproductive one with probability p_d. In either case we traverse some number $0 \le k \le r-1$ of unproductive sub-trees, each of which adds C steps. In case of a productive state, the number depends on the position of τ_1 in the set of $k+1$ elements:

$$\begin{aligned} h_{[H|T]}^{A_{i+1} \cup \{\bot\}} &= p_d \cdot \sum_{k=0}^{r-1} \binom{r-1}{k} p_d^{r-1-k}(1-p_d)^k \cdot kC && \text{(unproductive)} \\ &\quad + (1-p_d) \cdot \sum_{k=0}^{r-1} \binom{r-1}{k} \frac{p_d^{r-1-k}(1-p_d)^k}{k+1} && \text{(productive)} \\ &\quad \cdot \sum_{m=0}^{k} mC + h_i + q^{(i)}(m-k)C && \text{(proposition 2 and lemma 2)} \end{aligned}$$

Note the denominator $k+1$ accounts for the probability of placing τ_i at position $m = 0, 1, \dots, k$. At position m, there are mC unproductive sub-trees *before* reaching τ_1 and $(m-k)$ after. In the third line, we split the recursive hitting time using Lemma 2. Note that the exit state Exit_s of each recursive sub-tree root s reached in this way is either $\bot$ (if $k = m$) or the adjacent unproductive recursive sub-tree root (cf. Figure 3). By Proposition 2 Item 1 we may use recursion.

Computing the inner sum and canceling out the denominator $k+1$ from the second line, we get:

$$h_{[H|T]}^{A_{i+1} \cup \{\bot\}} = p_d \cdot \sum_{k=0}^{r-1} \binom{r-1}{k} p_d^{r-1-k}(1-p_d)^k \cdot kC$$

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

$$+ (1-p_d) \cdot \sum_{k=0}^{r-1} \binom{r-1}{k} p_d^{r-1-k} (1-p_d)^k \cdot \left(\frac{kC}{2} + h_i + q^{(i)} \frac{kC}{2} \right)$$
$$= p_d(1-p_d)(r-1)C + (1-p_d) \left(h_i + (1+q^{(i)}) \frac{(r-1)(1-p_d)C}{2} \right)$$

where the Binomial sums are computed as in the proof of Theorem 3. The formulas for the remaining states are completely analogous, but we have additional steps for the intermediate detours over []. It is thus convenient to express them in terms of $h_{[H|T]}^{A_{i+1} \cup \{\perp\}}$:

$$h_{[][H|T]}^{A_{i+1} \cup \{\perp\}} = 1 + h_{[H|T]}^{A_{i+1} \cup \{\perp\}}$$
$$h_{[H|T][]}^{A_{i+1} \cup \{\perp\}} = h_{[H|T]}^{A_{i+1} \cup \{\perp\}} + (1-p_d) q^{(i)} \qquad (*)$$

For the second identity, note that visits to [] are counted only in case of failure and are thus weighted by $q^{(i)}$.

We can now turn to h_{i+1}. By Eq. (1):

$$h_{i+1} = p_d^2 \cdot 0 + (1-p_d)p_d + (1-p_d)p_d \cdot h_{[H|T]}^{A_{i+1} \cup \{\perp\}}$$
$$+ \frac{(1-p_d)^2}{2} \cdot \left(h_{[][H|T]}^{A_{i+1} \cup \{\perp\}} + h_{[H|T][]}^{A_{i+1} \cup \{\perp\}} \right)$$

which, after substitution of $(*)$ and straightforward simplification becomes

$$h_{i+1} = \frac{1}{2} \left(1 - p_d^2 + (1-p_d)^3 q^{(i)} \right) + (1-p_d) h_{[H|T]}^{A_{i+1} \cup \{\perp\}}$$

We substitute $h_{[H|T]}^{A_{i+1}}$ and Proposition 2 Item 2, then simplify, isolating the terms that are dependent on i:

$$h_{i+1} = (1-p_d)^2 h_i - \frac{1 + C(r-1)}{2} \cdot (1-p_d)^{2i+4}$$
$$+ C(1-p_d)^2(r-1) + \frac{(1-p)(1+p) + (1-p)^3}{2}$$

With $\alpha = (1-p_d)^2$, $\beta = -\frac{1+C(r-1)}{2} \cdot (1-p_d)^4$, and $\gamma = C(1-p_d)^2(r-1) + \frac{(1-p)(1+p)+(1-p)^3}{2}$ we get the following formula (e.g. by iterative substitution):

$$h_{i+1} = \alpha^{i+1} h_0 + (i+1)\beta\alpha^i + \gamma \frac{1-\alpha^{i+1}}{1-\alpha} \qquad (8)$$

Its correctness is easily shown by induction on $i \geq 0$.

Substituting h_0, α, β, and γ into Eq. (8) gives:

$$h_i = (1-p_d)^{2i} h_0 - i \cdot (1-p_d)^{2(i-1)+4} \cdot \frac{1 + C(r-1)}{2}$$
$$+ \left(C(1-p_d)^2(r-1) + \frac{(1-p)(1+p) + (1-p)^3}{2} \right) \frac{1 - (1-p_d)^{2i}}{1 - (1-p_d)^2}$$

which simplifies to the desired formula. □

Substituting into Eq. (7) gives the following theorem:

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Theorem 4.
Let $p_d \in (1-\frac{1}{\sqrt{r}}, 1)$, $r \in \mathbb{N}$. Denote by C the constant from Theorem 3. Let τ be a test case of length l and $A_\tau \subseteq \mathcal{S}$ the set of states that output τ. Then:

$$h_\varepsilon^{A_\tau} = \frac{1 + \left(\frac{C(1-p_d)^2(r-1)}{p_d(2-p_d)} + \frac{(1-p)(1+p)+(1-p)^3}{2p_d(2-p_d)}\right)}{(1-p_d)^{2l+1}} + \frac{(1+p_d)(1+C)}{2} - \frac{C(1-p_d)(r-1)}{p_d(2-p_d)} - \frac{(1+p_d)+(1-p_d)^2}{2p_d(2-p_d)} - l\cdot(1-p_d)\cdot\frac{1+C(r-1)}{2} - (1-p_d) + \frac{1}{(1-p_d)}$$

All that remains is to prove Lemma 2:

Proof of Lemma 2.

$$h_x^A = \sum_{n=0}^{\infty} n \cdot \Pr[H_x^A = n, H_x^A < H_x^y] + \sum_{n=0}^{\infty}\sum_{k=0}^{n-1} n \cdot \Pr[H_x^A = n, H_x^A > H_x^y, H_x^y = k]$$

Using Fubini's theorem, the second sum becomes

$$\sum_{k=0}^{\infty} \Pr[H_x^y = k, H_x^A > H_x^y] \sum_{n=k+1}^{\infty} n \cdot \Pr[H_x^A = n | H_x^y = k, H_x^A > H_x^y]$$
$$= \sum_{k=0}^{\infty} \Pr[H_x^y = k, H_x^A > H_x^y] \sum_{m=1}^{\infty} (m+k) \Pr[H_y^A = m]$$

and since $\Pr[H_y^A = 0]$, because $y \notin A$, and moreover $\mathcal{M}$ is recurrent (so $\Pr[H_y^A < \infty] = 1$) this is equal to

$$\sum_{k=0}^{\infty} \Pr[H_x^y = k, H_x^A > H_x^y] \left(k + h_y^A\right)$$
$$= \left(\sum_{k=0}^{\infty} \Pr[H_x^y = k, H_x^A > H_x^y] k\right) + h_y^A \cdot \Pr[H_x^A > H_x^y].$$

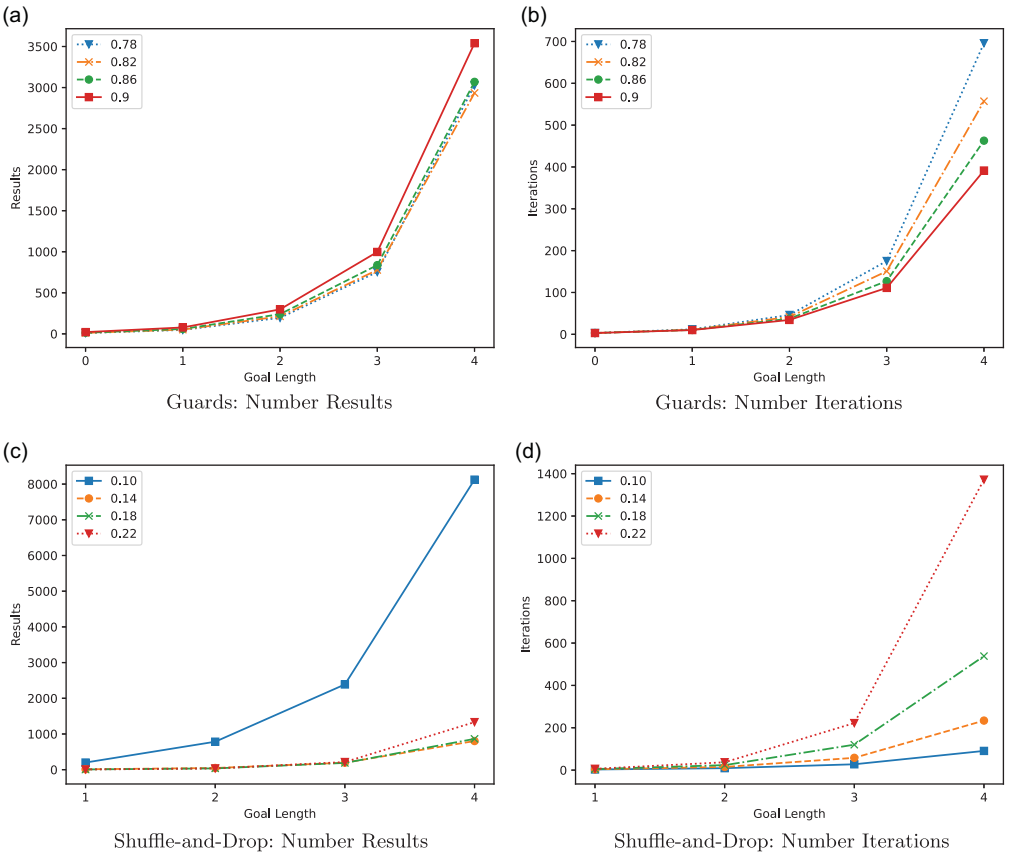
Because $y \notin A$, we have $H_x^A \neq H_x^y$ and so $\Pr[H_x^y = k, H_x^A > H_x^y] + \Pr[H_x^A = k, H_x^A < H_x^y] = \Pr[H_x^{A\cup\{y\}} = k]$. Hence the remaining two infinite sums add up to $h_x^{A\cup\{y\}}$. □

4 Evaluation

In the following section, we empirically evaluate the randomization approaches outlined in section 3. We implement the strategy via *guards* using SWI-Prolog (Wielemaker *et al.* 2012). To implement the *drop-and-shuffle* strategy, we choose Go-Prolog ichiban/Prolog (2024). Go-Prolog has a small and easily modifiable code-base, which simplifies experiments of this kind. We benchmark these approaches with various choices for the configurable probabilities.

For the benchmarks, we use two programs: a slightly altered version of the program from listing 1 and a program generating basic arithmetic expressions, shown in listing

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press



Guards: Number Results

Guards: Number Iterations

Shuffle-and-Drop: Number Results

Shuffle-and-Drop: Number Iterations

Fig. 4. Iterations and results until test case `[second, ..., second]` is reached.

3. In both cases, we count the number of *iterations*, which is defined as the number of times the program needs to be re-run to obtain the target result, and the *results*, which is the number of outputs of the program before we obtain the target result.

4.1 Benchmark 1: Command sequences

The first benchmark is based on listing 1 with two adjustments We limited the number of available commands to three. Moreover, each `command/1` predicate simply unifies its argument with a corresponding constant (in our case `first`, `second`, and `third`). We executed each benchmarks 1000 times. The results are shown in Figure 4.

The *goal length* each benchmark lists on its x-axis is the length of a list consisting solely of the constant symbol `second` the respective number of times. This guarantees that we would not find this test case with the standard depth-first, left-first search behavior, but also that is not the path that would be picked last with depth-first search. For our implementation, we relied on Janus Andersen and Swift (2023) for SWI as the Python-Prolog bridge to gather the results.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Guard-Approach Benchmarks: Every `command/1` predicate had an equal probability of $\frac{1}{3}$ for the steady probability. The different plots mark different continuation probabilities p_c used for the respective queries.

Figure 4a shows the number of results until a specific target test case is found. As expected, the number of results drastically increases with the target list size. Further, only a continuation probability of 0.9 has a higher number of results. Figure 4b shows how many iterations were necessary until the determined target was found. Similar to the number of results, the number of iterations also grows with increasing list size of the expected outcome. As the continuation probability increases the total number of iterations decreases.

Drop-and-Shuffle Benchmarks: As described above, for the Go-Prolog variant we implemented a drop-probability as discussed in section 3. Otherwise, the benchmarks are still conducted using the same pattern as described above for increasing goal lengths. Figure 4c shows the number of produced results whereas Figure 4d shows the number of iterations.

Note that dropping a clause with probability 0.1 is the same as proceeding to explore it with probability 0.9. Hence, the probabilities in Figures 4c and 4d are dual to those above. But note that in this way they *do not fulfill the premise of Theorem 3*: They are below $1 - \frac{1}{\sqrt{r}} \approx 0.423$. The mean hitting time C is thus infinite. And yet, we obtain results! This might seem paradoxical, but is due to the fact that we return to $\bot$ with probability 1, even if there is no finite mean. Indeed, this makes these measurements particularly interesting, because they have no defined mean to converge to – the law of large numbers applies only if the distribution has finite mean!

Note that the drop-and-shuffle randomization strategy is much more coarse than the guard strategy by design: The 0.1 drop probability applies to *both* the `t` predicate as well as the `command{1,2,3}` predicates. This is quite different from the previous scenario, where $p_c = 0.9 \gg p_1 = \cdots = p_r = 0.33$ were distinct. Consequently, both the number of iterations and the number of results are notably higher for the drop-and-shuffle approach: A probability of 0.1 to drop a clause produces significantly more solutions until a specified goal is found. The drop probabilities of 0.14 and 0.18 are rather similar for all specified goal lengths. On the other hand, the number of iterations signals that the number of iterations rises with a higher dropping probability. In Figure 4c, the probability 0.14 outperforms both 0.10 and 0.18.

In Gelderie *et al*. (2024), we conjectured that this is due to an inflection point. But the results of section 3.2 now show that this reasoning is not verifiable: An undefined function cannot have an inflection point. It seems impossible to know whether this is an artifact of the inherent randomness of the measurements (that, we know, cannot converge to a non-existent mean), or some other effect. In any case, it underscores that the drop-and-shuffle approach is unwieldy and difficult to analyze.

4.2 Benchmark 2: Arithmetic expressions

In this section, we provide the results of another benchmark to showcase the behavior of both approaches in a more complicated setting. This time, our goal is to show the behavior of the two randomization approaches in settings other than those considered in the previous sections. We use the program in listing 3, which generates basic arithmetic

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

```
1  expr(X) :- const(X).
2  expr([Operator, Operands]) :- unpack(Operator, Operands).
3
4  const(1).
5  const(2).
6  const(3).
7
8  unpack(plus, [A, B]) :- expr(A), expr(B).
9  unpack(times, [A, B]) :- expr(A), expr(B).
10 unpack(minus, [A]) :- expr(A).
```

Listing. 3. A program generating arithmetic expressions.

Table 1. *Example expressions of smallest possible size for the target values (in Polish notation)*

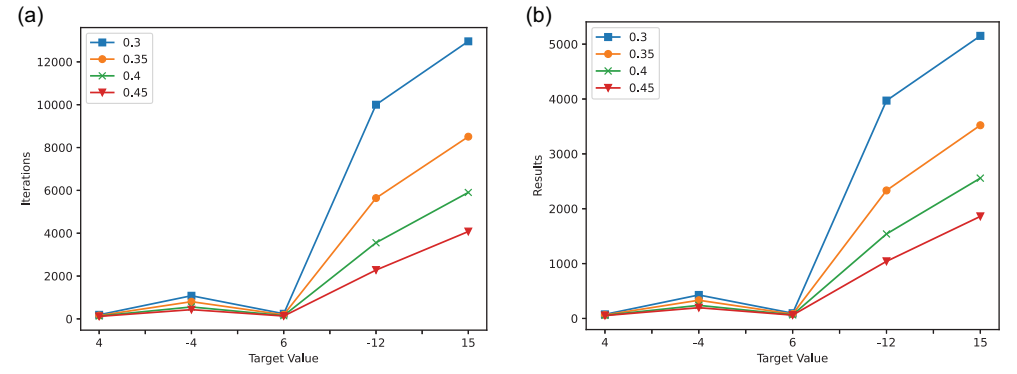
Target Value	Shortest Expression
4	`+(1,3)`
−4	−`(+(1,3))`
6	×`(2,3)`
−12	−(×`(3,+(2,2))`)
15	×`(3,+(2,3))`

expressions build from the two binary operators `+` and ×, as well as the unary operator −. Note that only the numbers 1, 2, and 3 can be used within an expression. Expressions have the form `[minus, [plus, [1,3]]]`. In the text below, we use the more readable symbolic representation −`(+(1,3))` (Polish notation). The *value* of the previous expression is −4.

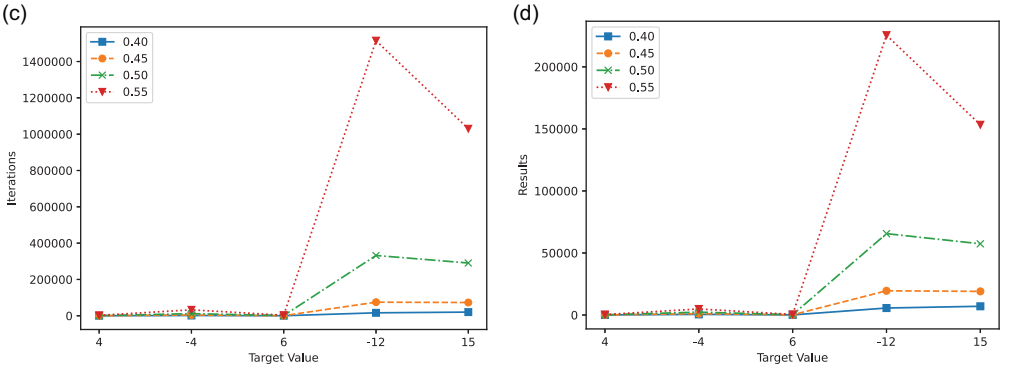
Our benchmark counts the iterations and number of results (defined as above in section 4.1) until an expression that evaluates to a specific *target value* is found. The target values, along with example expressions of shortest length, are provided in Table 1. As before, we repeat our experiment 1000 times. Clearly, for any target integer value x, there are infinitely many expressions that evaluate to x. However, some values can be reached with relatively simple expressions, whereas others require more complex, nested expressions that can only be found at lower depths in the SLD-tree.

Note that the results from the preceding sections do not directly extend to the program that we study here: The program is different (though similar in structure), and we now investigate the number of steps needed to reach *any* output from an *infinite set* of target outputs. The second point, in particular, is a major difference from the previous setting. Consider, for example, the target value 15. Expressions with value 15 are ubiquitous. Suppose during resolution, we arrive at a partially expanded expression `+(a,X)`, where `X` is yet to be derived and where the value of `a` is an *arbitrary* integer. Then there are infinitely many expressions for `X` that will give the value 15. The only partial expression that cannot ever be completed to a full expression evaluating to 15 is of the form ×`(a,X)` for an expression `a` with a value that is not a divisor of 15. This means, that we have a much larger probability of arriving at a "desired" output than before.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press



Guard: Number of iterations Guard: Number of results



Shuffle-and-Drop: Number of iterations Shuffle-and-Drop: Number of results

Fig. 5. Iterations and results until the generated expression has a specific value.

Guard-Approach Benchmarks: For the guard approach, we used the continuation probabilities $p_c = 0.3$ to $p_c = 0.45$ in increments of 0.05. The continuation probability p_c was used only for the clause in line 2 of listing 3. All other clauses, including the facts in lines 4–6, were guarded with a fixed probability of 0.33.

The results of the guard-approach benchmark is shown in Figures 5a and 5b. The first observation is that lower continuation-probabilities result in a larger number of results until a target clause is reached. This is in contrast to the previous benchmark (cf. section 4.1), where higher probabilities lead to a large number of results. This is likely because we now try reach any one expression from an infinite set of expressions. If the continuation probability is higher, we reach deeper into the SLD tree. Unlike before, however, we are quite likely to find our target that way.

Note that expressions with a negative sign require both more iterations and produce more unwanted results, before being reached. Target values requiring more complex expressions take longer to reach, as expected. There is significant increase in the time required to produce a target result, if three or more sub-expressions are needed (i.e. for -12 and 15). Note also that there is a small peak for expression -4. This is likely, because -4 requires on additional operator if compared with $4 = 2 + 2$ and $6 = 2 \cdot 3$.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Drop-and-Shuffle Benchmarks: We used a drop-probability p ranging from 0.4 to 0.55 (in increments of 0.05). Values of $p > 0.55$ took excessively longer to benchmark, and we could not complete 1000 test-runs within four days of running the benchmark for such values of p. The results of the benchmarks are shown in Figures 5c and 5d.

First, we note that the numbers are orders of magnitude larger than for the guard approach. While the numbers are difficult to compare (unlike section 4.1 probabilities are not completely dual), there appears to be a significant increase in runtime and uninteresting results that do not evaluate to the target value. As we have noted before, the shuffle-and-drop strategy is more coarse and the drop probability affects all clauses, not just the one in line 2 of listing 3. This is the most likely explanation for the excessive runtime.

Next, note that the metrics for target value 15 are consistently lower than for target value −12. We do not observe this effect for the guard approach in Figures 5a and 5b. In absence of mathematical rigor, we can, again, only conjecture as to the cause. There are two ways of reading this result: It shows that 15 is intrinsically easier to reach in the shuffle-and-drop approach than −12, or that −12 is intrinsically harder to reach, when using the shuffle-and-drop approach. We conjecture that the second interpretation is correct. The lowest test case that can produce −12 is at a greater depth than 15 (see Table 1). Moreover, the number −12 it requires even greater depth to reach if the outermost operator is not '−'. Looking back to Theorem 4 in subsection 3.2.2, we see that the runtime grows exponentially in *twice* the depth of the test case. This result does not extend to our present setting, but it seems likely that the runtime must grow at least exponentially in the depth of the *shortest* possible derivation of the desired output, governed by a similar growth-function to that given in Theorem 4. If that is the case, reaching −12 becomes much more difficult than reaching 15. While a similar argument would seem to apply to the guard approach as well, where we don't see this effect, the base of the exponential is smaller and the exponent is not scaled by two (see Corollary 2). It is quite possible that the relatively short expressions yielding −12 are dominated by other factors in the guard approach setting, due to the smaller base and exponent.

5 Conclusion

We have presented two approaches to randomize the SLD derivation of test cases in Prolog and studied their performance in terms of expected time to hit a test case, and mean number of test cases produced. To this end, we presented a detailed analysis of the random behavior of test-case generation using Prolog and Markov chains. Our theorems allow a precise calibration of the probabilities to adjust the expected number of test cases per query. When looping on such a query, the rate of growth of the mean-hitting time for a given test case is exponential in its depth, where the base is the product of the involved probabilities. We then compared both strategies and various sets of values for the involved probabilities empirically. We find that the guard approach that uses an unmodified Prolog implementation provides a very fine-grained control over the randomization and thus produces test cases quicker.

In future work, we plan to study the semantics of this approach when negation-as-failure is involved. In particular, randomization may lead to a false refutation of

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

`q(t_1,...,t_k)` in the goal `\+ q(t_1,..., t_k)`. However, this may be acceptable, if it occurs with low probability. In a similar vein, the treatment of negation as failure might require randomization strategies entirely different from those we have presented here, which is another interesting topic for future research.

References

Andersen, C. and Swift, T. 2023. The janus system: a bridge to new prolog applications. In *Prolog: The Next 50 Years*. Springer, 93–104.

Bougé, L., Choquet, N., Fribourg, L. and Gaudel, M.-C. 1985. Application of prolog to test sets generation from algebraic specifications. In *International Joint Conference on Theory and Practice of Software Development*. Springer, 261–275.

Casso, I., Morales, J. F., López-García, P. and Hermenegildo, M. V. 2019. An integrated approach to assertion-based random testing in prolog. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 159–176.

Denney, R. 1991. Test-case generation from prolog-based specifications. *IEEE Software* 8, 2, 49–57.

Dewey, K., Roesch, J. and Hardekopf, B. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14. Association for Computing Machinery, New York, NY, USA, 725–730.

Duran, J. W. and Ntafos, S. C. 1984. An evaluation of random testing. *IEEE Transactions on Software Engineering* SE-10, 4, 438–444.

Gelderie, M., Luff, M. and Peltzer, M. 2024. Impact and performance of randomized test-generation using prolog. In *Logic-Based Program Synthesis and Transformation: 34th International Symposium, LOPSTR 2024, Milan, Italy, September 9-10, 2024, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 166–182.

Gorlick, M. M., Kesselman, C. F., Marotta, D. A. and Parker, D. S. 1990. Mockingbird: a logical methodology for testing. *The Journal of Logic Programming* 8, 1-2, 95–119.

Gu, Q. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2201–2203.

Hoffman, D. M. and Strooper, P. 1991. Automated module testing in prolog. *IEEE Transactions on Software Engineering* 17, 9, 934.

Ichiban/Prolog. 2024. ichiban/prolog. Visited: 2024-05-03.

Ince, D. C. 1987. The automatic generation of test data. *The Computer Journal* 30, 1, 63–69.

Kim, Y. G., Hong, H. S., Bae, D.-H. and Cha, S. D. 1999. Test cases generation from uml state diagrams. *IEE Proceedings - Software* 146, 4, 187–192.

Meyer, B., Ciupa, I., Leitner, A. and Liu, L. L. 2007. Automatic testing of object-oriented software. In *SOFSEM 2007: Theory and Practice of Computer Science*, van Leeuwen, J., Italiano, G. F., van der Hoek, W., Meinel, C., Sack, H. and Plášil, F. Eds. Springer, Berlin, Heidelberg, Berlin Heidelberg, 114–129.

Miller, B. P., Fredriksen, L. and So, B. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM* 33, 12, 32–44.

Miller, E. F. and Melton, R. A. 1975. Automated generation of testcase datasets. In *Proceedings of the International Conference on Reliable Software*. Association for Computing Machinery, New York, NY, USA, 51–58.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press

Norris, J. 1998. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, New York, NY, USA.

Pesch, H., Schnupp, P., Schaller, H. and Spirk, A. P. 1985. Test case generation using prolog. In *Proceedings of the 8th International Conference on Software Engineering*, 252–258.

Ramler, R., Winkler, D. and Schmidt, M. 2012. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *2012, 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 286–293.

Siddiq, M. L., Da Silva Santos, J. C., Tanvir, R. H., Ulfat, N., Al Rifat, F. and Carvalho Lopes, V. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24. Association for Computing Machinery, New York, NY, USA, 313–322.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-prolog. *Theory and Practice of Logic Programming* 12, 67–96.

Xu, F. F., Vasilescu, B. and Neubig, G. 2022. In-ide code generation from natural language: promise and challenges. *ACM Transactions on Software Engineering and Methodology* 31, 1.

Zech, P., Felderer, M. and Breu, R. 2019. Knowledge-based security testing of web applications by logic programming. *International Journal on Software Tools for Technology Transfer* 21, 221–246.

Zech, P., Felderer, M. and Breu, R. 2013. Security risk analysis by logic programming. In *Risk Assessment and Risk-Driven Testing: First International Workshop, RISK 2013, Held in Conjunction with ICTSS 2013, Istanbul, Turkey, November 12, 2013. Revised Selected Papers 1*. Springer, 38–48.

Zeng, Z., Ciesielski, M. and Rouzeyre, B. 2002. Functional test generation using constraint logic programming. In *SOC Design Methodologies: IFIP TC10/WG10. 5 Eleventh International Conference on Very Large Scale Integration of Systems-on-Chip (VLSI-SOC'01) December 3-5, 2001, Montpellier, France*. Springer, 375–387.

https://doi.org/10.1017/S1471068425100082 Published online by Cambridge University Press