Impact and Performance of Randomized Test-Generation using Prolog* **

Marcus Gelderie[1][0009−0003−0291−3911], Maximilian Luff[1], and Maximilian Peltzer[1]

Aalen University of Applied Sciences
Beethovenstr. 1, 73430 Aalen, Germany
{firstname.lastname}@hs-aalen.de
https://www.hs-aalen.de

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 inter-dependent 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. [15,11,17,13]). 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 [12], based on natural language [21], and, more recently, using large-language models [8,19]. But tests have also been

* This work was created as part of a project funded by the German Federal Ministry of Education and Research under grant number 16KIS193K. The authors are responsible for the content of this publication.

generated according to formal or semi-formal specifications [24,5]. Particularly when formal methods are used, one often has to deal with a very large, even infinite, 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. [18,14,6,18,14,3]).

Prolog is a natural fit to generate test-cases that follow a logical pattern (as opposed to unstructured testing, as is done, for example, in many forms of Fuzzing [14]). Generating test-cases using Prolog has been studied in the past [17,9,3,4]. It has been applied to software-testing in general, but also to specialized areas, such as security testing [23,5]. Some approaches also use randomization to explore the space of test-cases [3]. 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 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 a 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. 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 [3], 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.

In the following we study the effects on randomizing the resolution in this way. Due to space constraints, we focus on the first proposed randomization strategy (though the analysis is almost identical in both cases). We give a detailed

description of the resulting Markov chain and analyze its probability structure. We show that, provided the guards are chosen appropriately, the number of test cases produced is finite and given by a simple equation in terms of the guard probabilities. 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.

Finally, we study the randomization procedures from an empirical perspective and provide comparisons between the two aforementioned approaches. We implement the *guards approach* to randomization in SWI-Prolog [20] and the *drop-and-shuffle* approach in Go-Prolog [10]. We chose Go-Prolog for its accessible 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.

Related Work Some early works on test-case generation using Prolog are [17,2,9,4]. Automated test-case generation in Prolog was described by Pesch and Schaller [17]. 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 [9]. The authors automated the generation procedure of tests for modules written in C with Prolog.

Bougé et al. [2] 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.

Richard Denney [4] 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 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. [7] also introduce a methodology for formal specifications. For this task, they use constraint logic programming to describe the system under test's behaviour. 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. [3] 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. [22,23] 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 as was also used in Fuzzing [5] by Dewey et al. 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 $l \in \mathbb{N}_0$. 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, e.g. [16]. 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

[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.

an infinite sequence of random variables $X_n \in \mathcal{S}$. The triple $((X_n)_{n\in\mathbb{N}_0}, p, \mathsf{Init})$ 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 $((X_n)_{n\in\mathbb{N}}, p, s)$. 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 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 $((X_n)_{n\in\mathbb{N}_0}, p, s)$ (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 $< 700\,\text{ms}$").

At a very abstract level, such a sequence of test-inputs could be generated with the Prolog program shown in listing 1.1. All valid substitutions for `X` in the query `t(X)` are input sequences to our fictional system. Since this program will

Listing 1.1: A program generating randomized sequences of test inputs.

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

Listing 1.2: Guard clauses.

```
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, /* ... */ .
% ...
```

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 1.2. Note that the symbols `p_cont`, `p_1`,... are meant to represent float constants between 0 and 1, and can be adjusted as needed. In effect, some subtrees of the SLD-tree are then randomly left unexplored.

A similar approach was proposed by Casso et. al. in [3]. 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). Results regarding hitting times require a more involved variant of the analysis given in section 3.2. Due to space constraints, we cannot elaborate here.

3.1 Number of Generated Tests

The program $\mathcal{P}$ shown in listings 1.1 and 1.2 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?

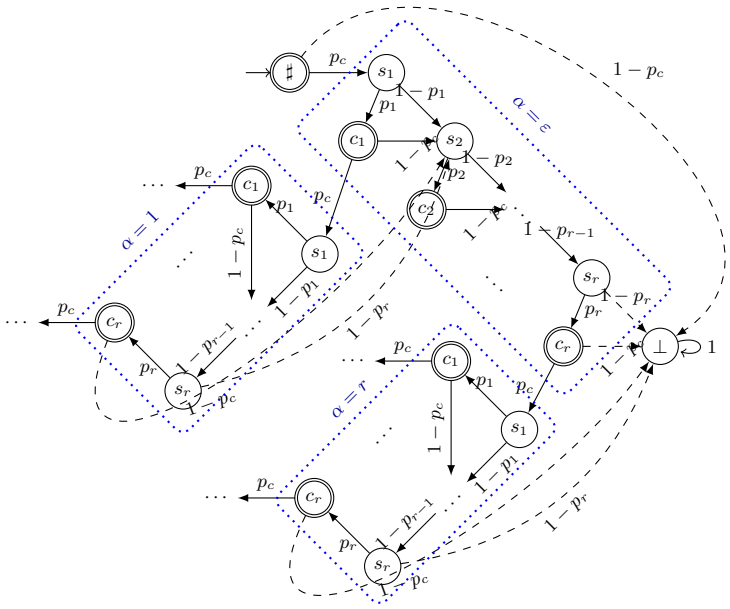


Fig. 1: The Markov chain corresponding to $\mathcal{P}$ with $\mathsf{Init} = \sharp$ and *blocks* $\alpha \in \{1, \ldots, r\}^*$ surrounded by blue boxes.

The 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 fig. 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 $\sharp$ corresponds to the empty list. $\bot$ is the only absorbing state. It corresponds a termination of the resolution algorithm.

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, $\alpha \in \{1, \dots, r\}^* \} \cup \{\bot, \sharp\}$ and transition probabilities $p(s, s')$ for $s, s' \in \mathcal{S}$:

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

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

This motivates the following transition probabilities

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

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 fig. 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 $s_0 s_1 s_2 \cdots$ 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 subtree 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 subtree 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:*

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

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 $\eta \stackrel{def}{=} r \cdot p_{\mathsf{max}} \cdot p_c < 1$, 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 $\Pr[I_\beta = 1] \leq (p_{\mathsf{max}} \cdot p_c)^l$ for all β. This gives (using linearity of expectation and that I_β is independent from X_β for all β):

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

□

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 $O(s) \leq N(s)$.

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

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

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; i.e. s_1 is understood to mean s_1^α.

Every path from s_1 to A must visit $s_2, \dots, s_r$. 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)

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

Solving for C proves the theorem. The proof for $\mathsf{E}[O(s_1)]$ is similar. □

Note that for any given state s_1^α, the mean hitting time $h^{s_1^\alpha}_\sharp = \sum_{n \geq 0} \Pr[H^{s_1^\alpha}_\sharp \geq n] \geq \sum_{n \geq 0} 1 - p_c = \infty$ (where we use $\mathsf{E}[X] = \sum_{n \geq 0} \Pr[X \geq n]$ for any random variable that 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.

3.2 Infinite Looping and Time-To-Hit

As shown in lemma 1, the program in listing 1.1 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 $\alpha_j = r$ $(1 \le j \le t)$.

With this, we may compute:

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

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 $\alpha = \beta \cdot j$.

$$\begin{aligned} h^{s_1^\alpha}_{\sharp} =& h^{s_j^\beta}_{\sharp} + 1 \\ &+ (1-p)(h^{\sharp}_{s_{j+1}^\beta} + h^{s_i^\alpha}_{\sharp}) && \text{(fall through to } s^\beta_{j+1}\text{)} \\ &+ p(1 + (1-p_c)(h^{\sharp}_{s_{j+1}^\beta} + h^{s_i^\alpha}_{\sharp})) && \text{(no visit to next block } \alpha\text{)} \\ =& h^{s_j^\beta}_{\sharp} + 1 + p + (h^{\sharp}_{s_{j+1}^\beta} + h^{s_i^\alpha}_{\sharp})(1-pp_c) \end{aligned}$$

This gives

$$h^{s_1^\alpha}_{\sharp} = \frac{h^{s_j^\beta}_{\sharp} + 1 + p + (1-pp_c) h^{\sharp}_{s_{j+1}^\beta}}{pp_c}$$

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

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

The following theorem gives a closed formula:

Theorem 2. *Let s_i^α for some $\alpha = \alpha_1 \cdots \alpha_t$. Let $\nu = pp_c$ and $\nu \cdot r < 1$. Then*

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

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

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):

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

□

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 $h_\sharp^{s_i^\alpha} \in \Omega(\nu^{-t})$.

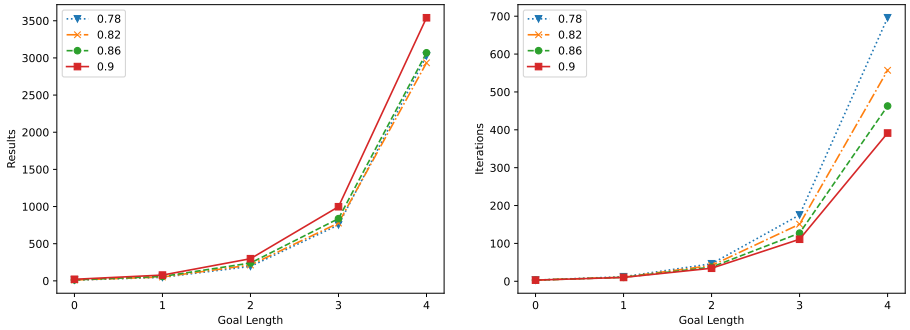
Choose suitable largest values $B \geq B_k$ for all $t \in \mathbb{N}$, $1 \leq k \leq t$, and $D \geq D_s$ for all $1 \leq s \leq t$. Bound eq. (6) from above by

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

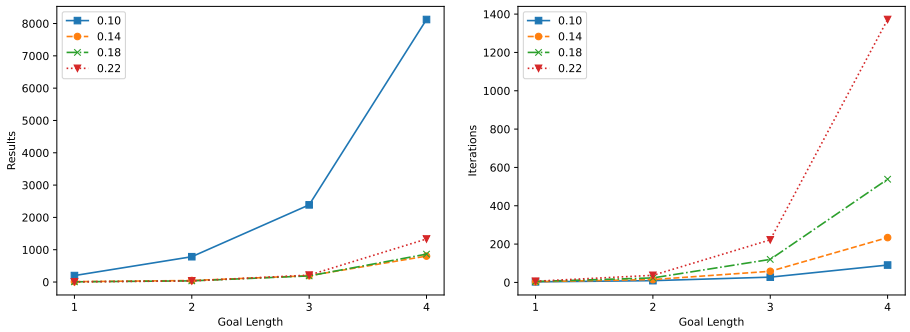
A well-known calculation via derivatives gives $\sum_{k=1}^{t} k \cdot \nu^k = \nu \sum_{k=1}^{t} k\nu^{k-1} \leq \nu \sum_{k=1}^{\infty} k\nu^{k-1} = \nu \cdot \frac{\mathrm{d}}{\mathrm{d}\nu} \sum_{k=0}^{\infty} \nu^k = \frac{\nu}{(1-\nu)^2}$. With that we have

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

□



(a) Guards: Number Results (b) Guards: Number Iterations



(c) Random Select: Number Results (d) Random Select: Number Iterations

Fig. 2: Iterations and results until test-case `[second, ..., second]` is reached.

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 [20]. To modify the *shuffle-and-drop* strategy, we choose Go-Prolog [10] 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. All the benchmarks were done using a slightly altered version of the program from listing 1.1: 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 fig. 2.

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 behaviour, but also that is not the path that would be picked last with depth-first search. For our implementation, we relied on Janus [1] for SWI as the Python-Prolog bridge to gather the results.

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 2a 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 2b 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.

Shuffle-and-Drop 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 2c shows the number of produced results whereas fig. 2d 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 figs. 2c and 2d are dual to those above. However, the shuffle-and-drop 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 shuffle-and-drop 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 fig. 2c, the probability 0.14 outperforms both 0.10 and 0.18, hinting at an inflection point. This is likely due to the dual role of the dropping probability, which governs the exploration of a specific test *and* the probability of exploring the SLD-tree below it: Too high, and the number of tests-until-target increases; too low, and the target-test is skipped at the target depth.

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 using 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 `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 entirely different randomization strategies than we have presented here, which is another interesting topic for future research.

References

1. Andersen, C., Swift, T.: The janus system: a bridge to new prolog applications. In: Prolog: The Next 50 Years, pp. 93–104. Springer (2023)
2. Bougé, L., Choquet, N., Fribourg, L., Gaudel, M.C.: Application of prolog to test sets generation from algebraic specifications. In: International Joint Conference on Theory and Practice of Software Development. pp. 261–275. Springer (1985)
3. Casso, I., Morales, J.F., López-García, P., Hermenegildo, M.V.: An integrated approach to assertion-based random testing in prolog. In: International Symposium on Logic-Based Program Synthesis and Transformation. pp. 159–176. Springer (2019)
4. Denney, R.: Test-case generation from prolog-based specifications. IEEE Software **8**(2), 49–57 (1991)
5. Dewey, K., Roesch, J., Hardekopf, B.: Language fuzzing using constraint logic programming. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 725–730. ASE '14, Association for Computing Machinery, New York, NY, USA (2014). `https://doi.org/10.1145/2642937.2642963`
6. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE transactions on Software Engineering (4), 438–444 (1984)
7. Gorlick, M.M., Kesselman, C.F., Marotta, D.A., Parker, D.S.: Mockingbird: a logical methodology for testing. The Journal of Logic Programming **8**(1-2), 95–119 (1990)
8. Gu, Q.: 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. pp. 2201–2203 (2023)
9. Hoffman, D.M., Strooper, P.: Automated module testing in prolog. IEEE Transactions on Software Engineering **17**(9), 934 (1991)
10. ichiban/prolog: ichiban/prolog, `https://github.com/ichiban/prolog`, visited: 2024-05-03
11. Ince, D.C.: The Automatic Generation of Test Data. The Computer Journal **30**(1), 63–69 (01 1987). `https://doi.org/10.1093/comjnl/30.1.63`
12. Kim, Y.G., Hong, H.S., Bae, D.H., Cha, S.D.: Test cases generation from uml state diagrams. IEE Proceedings-Software **146**(4), 187–192 (1999)
13. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic testing of object-oriented software. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007: Theory and Practice of Computer Science. pp. 114–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

14. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (dec 1990). https://doi.org/10.1145/96267.96279
15. Miller Jr, E.F., Melton, R.A.: Automated generation of testcase datasets. In: Proceedings of the international conference on Reliable software. pp. 51–58 (1975)
16. Norris, J.: Markov Chains. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press (1998)
17. Pesch, H., Schnupp, P., Schaller, H., Spirk, A.P.: Test case generation using prolog. In: Proceedings of the 8th international conference on Software engineering. pp. 252–258 (1985)
18. Ramler, R., Winkler, D., Schmidt, M.: 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. pp. 286–293. IEEE (2012)
19. Siddiq, M.L., Santos, J., Tanvir, R.H., Ulfat, N., Rifat, F.A., Lopes, V.C.: Exploring the effectiveness of large language models in generating unit tests. arXiv preprint arXiv:2305.00418 (2023)
20. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)
21. Xu, F.F., Vasilescu, B., Neubig, G.: In-ide code generation from natural language: Promise and challenges. ACM Trans. Softw. Eng. Methodol. **31**(2) (mar 2022). https://doi.org/10.1145/3487569
22. Zech, P., Felderer, M., Breu, R.: 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. pp. 38–48. Springer (2014)
23. Zech, P., Felderer, M., Breu, R.: Knowledge-based security testing of web applications by logic programming. International Journal on Software Tools for Technology Transfer **21**, 221–246 (2019)
24. Zeng, Z., Ciesielski, M., Rouzeyre, B.: 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. pp. 375–387. Springer (2002)