Performance Fuzzing with Reinforcement-Learning and Well-Defined Constraints for the B Method

Jannik Dunkelau (✉) and Michael Leuschel

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstraße 1, 40225 Düsseldorf, Germany
jannik.dunkelau@hhu.de, michael.leuschel@hhu.de

Abstract. The B method is a formal method supported by a variety of tools. Those tools, like any complex piece of software, may suffer from performance issues and vulnerabilities, especially for potentially undiscovered, pathological cases. To find such cases and assess their performance impacts within a single tool, we leverage the performance fuzzing algorithm BanditFuzz for the constraint solving backends of the ProB model checker. BanditFuzz utilises two multi-armed bandits to generate and mutate benchmark inputs for the ProB backends in a targeted manner. We describe how we adapted BanditFuzz for the B method, which differences exist to the original implementation for the SMT-LIB standard, and how we ensure well-definedness of the randomly generated benchmarks. Our experiments successfully uncovered performance issues in specific backends and even external tooling, providing valuable insights into areas which required improvement.

Keywords: B method · ProB · Constraint solving · Performance Fuzzing · Reinforcement Learning · Multi-armed bandits

1 Introduction

The B method is a formal method for software [1] and systems development [2] as well as data validation [9]. Specifications are written in the B language, which is rooted in predicate logic, arithmetic and set theory. The B method is based on a correct-by-construction approach and refinement calculus [5,6].

ProB [24,25] is a model checker, animator, and constraint solver for the B method. Its native constraint solver backend makes use of constraint logic programming over finite domains (CLP(FD)) [11], it provides other solver backends to SAT [32] and SMT [35,36]. While existing work already assessed strengths and weaknesses of these backends in a general scope [16], the proposed BanditFuzz approach by Scott et al. [38] for performance fuzzing promises a more targeted generation of examples where the backends' performances differ significantly.

The original BanditFuzz [37] finds targeted benchmark examples within the SMT-LIB language [8], trying to increase the performance margin between two sets of solvers. It relies on two reinforcement learning agents which control a

fuzzing-generator. This more targeted approach allows achieving pathological inputs for solvers which are concise yet cause significant performance issues and are valuable for finding and debugging edge-cases, as the BanditFuzz case study already proved [37].

In this work, we implement the BanditFuzz algorithms for the B method and apply it to a selection of ProB's constraint solving backends, namely the native CLP(FD) backend (with varying parameters), a translation to Z3 [35], as well as a natively implemented SMT backend based on CDCL(T) [36]. One particular difficulty of applying BanditFuzz to B as opposed to SMT-LIB is the notion of well-definedness, i.e. we have to take care that the fuzzer does not generate predicates whose logical meaning is not well-defined. Our contributions are

- a fuzzer generator for the B method which ensures well-definedness of the generated targets,
- a performance fuzzer based on BanditFuzz for the B method, and
- an evaluation of the applicability of the BanditFuzz algorithm outside the SMT-LIB domain.

The remainder of this paper is organised as follows. Section 2 provides an overview of ProB, performance fuzzing, and Thompson sampling. Section 3 then explains the general workings of the BanditFuzz algorithm while Section 4 describes the implementation of the algorithm for ProB and the resulting differences and challenges in contrast to the original SMT-LIB implementation. We summarise our experiments with the implemented performance fuzzer in Section 5. Section 6 explores related work concerned with performance fuzzing. We conclude the paper with a discussion of our results in Sections 7 and 8.

2 Background

2.1 ProB and the B Method

ProB [24, 25] is a model checker, animator, and constraint solver for the B method [1]. It is implemented in SICStus Prolog [10, 12] and natively makes use of constraint logic programming over finite domains (CLP(FD)) [11]. This constraint solver lies at the heart of ProB. For instance, it is used during model checking for verification of machine invariants and thus responsible for finding specification violations as well as calculating the follow-up states (i.e., animation). The constraint solver is further utilised for symbolic verification, program synthesis [34], or test case generation [23].

The CLP(FD) based constraint solver, further referred to as ProB CLP(FD), works by setting up possible domains for any variable then propagating constraints modelling relationships between those variables, successively reducing said domains. If any domain becomes empty, no satisfying solution can be found. Otherwise, when propagation terminates, satisfying solutions consist of simply assigning each variable to one of the remaining values within their respective domains.

ProB further provides a translation to the SMT-LIBv2 standard [8] with connection to the Z3 prover [15]. For the connection to Z3, two separate SMT-LIB translations are available. The initial implementation [19] matches B operators with their corresponding counterparts within SMT-LIB where possible. Constructs such as set comprehensions with no direct counterpart are translated as axiomatised formulae containing quantifiers, or are interpreted as forms of set comprehensions [36]. A new more constructive translation [36] uses Z3's lambda functions instead of quantifiers. ProB's Z3 backend runs both translations in parallel and reports the first found solution.

ProB also features a native implementation of conflict driven clause learning modulo theories [28], which we will refer to as ProB CDCL(T) [36]. This was implemented after observing the benefits of the Z3 integration in certain problem domains and solves problems regarding still untranslated or suboptimally translated constructs, as ProB CDCL(T) works natively with B as input languages.

2.2 Performance Fuzzing

Fuzzing was originally proposed by Miller et al. [29] who tested various Unix utility programs with randomly generated input and found inputs to crash 24 % of the programs under test (PUT). Nowadays, fuzzing is a well-established technique to expose possible vulnerabilities and malfunctions of a PUT by constructing a fuzz generator that samples inputs which might not reside in the PUT's expected input space [26, 27].

Fuzzing algorithms can be distinguished as blackbox or whitebox fuzzing [26]. Blackbox fuzzing considers the PUT as a blackbox and thus has only the observable outputs as feedback. Whitebox fuzzing on the other hand relies on knowledge of internals of the PUT, allowing the algorithm for instance to generate inputs that cover all possible execution paths. We can further differentiate between generation based or mutation based fuzzing [27]. In generation based fuzzing, the fuzzing algorithm randomly creates inputs based on a given model describing the form of expected inputs of the PUT. Mutation based fuzzing does not rely on such a model but rather takes a so called seed-input which it then subsequently mutates.

Performance fuzzing extends the application domain to finding pathological inputs over which the PUT loses performance [14]. While not focused on finding vulnerabilities, the acquisition of small inputs which produce large runtimes still allows to find performance bugs within the PUT [38].

2.3 Thompson Sampling

Thompson sampling [39] is an approach for choosing actions in the multi-armed bandit (MAB) problem [4, 33]. In the MAB problem, a slot machine with N independent arms (corresponding to available actions) and time steps $t = 1, 2, 3, \ldots$ are given. Playing an arm at time step t yields a random reward R_t which is

sampled from an unknown but fixed distribution. Within T time steps, the goal is to maximise the expected total reward

$$\mathbb{E}\left[\sum_{t=1}^{T} \mu_{i(t)}\right]$$

where $\mu_{i(t)}$ denotes the expected reward for arm i at step t based on previous observations [4]. For Bernoulli distributed rewards, Thompson sampling assumes priors are distributed by the Beta distribution, which has two shape parameters $\alpha > 0$ and $\beta > 0$. These priors are initialised for each arm to be $\text{Beta}(1, 1)$, i.e. $\alpha = \beta = 1$ [4]. The algorithm samples N random numbers $\theta_1, \ldots, \theta_N$, one for each arm, from their respective distributions $\text{Beta}(\alpha_i, \beta_i)$ and selects $i(t) = \arg\max_i \theta_i$ as the arm to play next [13]. After observing the resulting reward, the chosen arm's prior distribution is updated in the following way: If the reward was positive, increment α_i, else increment β_i. See Algorithm 1 for a summary of the approach.

Algorithm 1: Thompson Sampling for Bernoulli bandits (adapted from [4, 13]).

Require: Arms $i = 1, \ldots, N$ with priors $\alpha_i = 1, \beta_i = 1$
  for all $t \in 1, \ldots, T$ **do**
    For each arm i sample θ_i from $\text{Beta}(\alpha_i, \beta_i)$
    Play arm $\hat{\imath} = \arg\max_i \theta_i$
    Observe reward $r_t \in \{0, 1\}$
    $\alpha_i \leftarrow \alpha_i + r_t$, $\beta_i \leftarrow \beta_i + (1 - r_t)$
  end for

3 The BanditFuzz Algorithm

BanditFuzz is a performance fuzzing algorithm for SMT solvers based on multi-agent reinforcement learning (RL) as proposed by Scott et al. [37, 38]. The algorithm is composed of a fuzzing unit as well as two reinforcement learning agents based on Thompson sampling.

BanditFuzz takes as input a set of target solvers and a set of reference solvers, and produces a new input (benchmark) which maximises the performance margin between the two sets of solvers. The performance margin is hereby measured as the runtime difference between the slowest reference solver and fastest target solver.

BanditFuzz employs two Thompson sampling agents. The first agent (further referred to as inner agent) selects a mutation for the so far best found benchmark. In terms of MAB, the arms correspond to available grammatical language constructs which can be inserted into the given input to mutate it. The insertion happens by replacement of existing constructs to not continuously expand the benchmark's size. The second (outer) agent was introduced in an

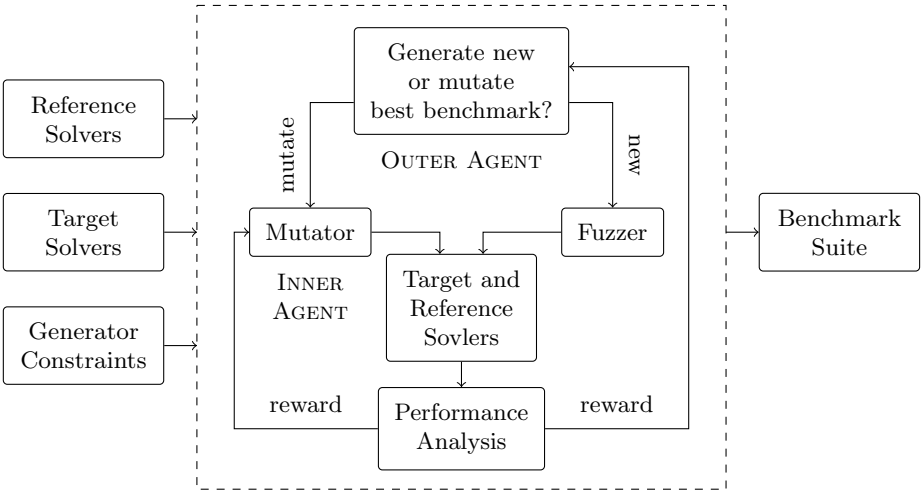

Fig. 1: Overview of the BanditFuzz architecture. Adapted from [38, Fig. 1]. The inner and outer RL agents to control whether to create a new input or how to mutate the current best input. Rewards are played back to the agents after performance analysis, based on whether the performance margin was increased.

updated version of BanditFuzz [38] and prevents the inner agent from becoming stuck in local extrema. Its action set consists of either choosing to mutate the best benchmark or to sample a new input via fuzzing. Figure 1 visualises this double-agent architecture.

After selecting a new benchmark, it is passed to each solver from both the reference and the target sets. If the resulting performance margin is now greater than what the previously best benchmark produced, a new best benchmark was found and a positive reward is played back to the agents, otherwise the reward is negative. The implementation contains a hyperparameter $\gamma \in [0, 1]$ which influences the mean decay during reward playback. This changes the Thompson sampling algorithm slightly, as shown in Algorithm 2. Note that α_i and β_i are now starting at 0 and that the distribution formula is changed to $\text{Beta}(\alpha_i + 1, \beta_i + 1)$.

Algorithm 2: Thompson Sampling for Bernoulli bandits with mean decay.

Require: Arms $i = 1, \ldots, N$ with priors $\alpha_i = 0, \beta_i = 0$; decay factor γ
  for all $t \in 1, \ldots, T$ **do**
    For each arm i sample θ_i from $\text{Beta}(\alpha_i + 1, \beta_i + 1)$
    Play arm $\hat{\imath} = \arg\max_i \theta_i$
    Observe reward $r_t \in \{0, 1\}$
    $\alpha_i \leftarrow \gamma\alpha_i + r_t$
    $\beta_i \leftarrow \gamma\beta_i + (1 - r_t)$
  end for

```
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x (* 4 y)) y))
(assert (>= (/ 1 x) 0))
(check-sat)
(get-value (x y))
```

(a) SMT-LIB

```
x : INTEGER & y : INTEGER
& x + 4*y = y
& 1/x >= 0
```

(b) Classical B

Fig. 2: Comparing SMT-LIB code to the respective Classical B formula for the constraint $x, y \in \mathbb{Z} \land x + 4y = y \land 1/x \geq 0$. The constraint is satisfied for $x = -3y \land x > 0$ and undefined for $x = y = 0$.

4 Adapting BanditFuzz for ProB

Implementing BanditFuzz for ProB results in major differences compared to the original implementation by Scott et al. [38]. The most obvious difference are the targeted formalisms, SMT-LIB [8] vs. Classical B [1]. This directly impacts the fuzz generator and mutator as these now have to account for valid B syntax. A direct result is the need to ensure well-definedness of the generated benchmarks as B implements well-definedness constraints over its operators [3, 22]. A less apparent distinction is the choice of implementation language. While the original BanditFuzz implementation in Python, the application to ProB in our case renders the use of SICStus Prolog [10] most sensible. This is due to ProB being implemented in SICStus Prolog itself, allowing us to take advantage of already existing tooling such as encoded typing information of available B syntax nodes (encoded as Prolog facts within ProB), ProB's well-definedness checker [22], or direct access to the API of ProB's respective constraint solving backends.

In the following, we summarize our Prolog-based implementation and highlight arising differences to the original BanditFuzz, the entailed challenges, and our solutions in more detail.

4.1 Fuzzing for Classical B instead of SMT-LIB

SMT-LIB [8] is a popular, standardised input language for SMT solvers and is supported by tools such as Z3 [15], CVC5 [7], Yices 2 [17], or Bitwuzla [30] among others. Syntactically, SMT-LIB has a Lisp-like syntax while B more closely resembles standard mathematical notation. For comparison, Figure 2 shows the same constraint in both, SMT-LIB and Classical B. As Lisp-dialects already resemble their respective abstract syntax trees (ASTs) and tend to be homoiconic, generating SMT-LIB code can be done in a top-down fashion without much hassle. For achieving the same luxury, and given that ProB is implemented in SICStus Prolog, we opted to write the fuzz generation in Prolog as well. Hence, we fuzz B ASTs directly instead of their textual formula representations.

4.2 Targeted Fuzzing and Mutating

We generate ASTs of B formulae in a top-down manner. Hereby, we ensure that the ASTs are complete trees with respect to a fixed, user specified depth to ensure generation of compact benchmarks. This means each leaf in the tree is at the specified depth counted from the root node. Further, each generated AST's root node is ensured to form a predicate (contrary to an expression which simply evaluates to a single value) which will be sent to the solver sets.

Initially, a random predicate is generated from which the BanditFuzz loop is started. Subsequently, the outer agent selects whether the current benchmark is to be mutated or replaced entirely. On replacement, a random new predicate is generated (and any resulting feedback is only propagated to the outer agent). When the benchmark is to be mutated, the inner agent selects a syntax node which has to be incorporated into the new AST. As B is a strongly typed language, there are two possibilities. First, there exists at least one similarly typed node in the current benchmark already. In that case, one of these nodes is selected at random and replaced by the new one. If possible, sub-nodes are reused. For cases where the new node has a higher arity or demands sub-nodes of different types, new sub-nodes are generated by the fuzzer on demand.

Second, if there exists no node of similar type in the current benchmark, a new AST is fuzzed which is guaranteed to contain the demanded node by first generating a sub-AST with the demanded node as root of size less or equal to the user-specified maximum, then generating parents and siblings until the required maximum height is reached. This approach diverges from the original BanditFuzz which did not guarantee the node selected by the inner agent to be part of the newly generated benchmark in cases where replacement was not possible.

4.3 Well-defined ASTs

Classical B implements the concept of well-definedness (WD) [3,22]. For instance, the B formula in Figure 2b is not well-defined as division by zero is possible within the subformula `1/x >= 1`. The B method consequently generates the WD proof obligation $\mathtt{x} \neq 0$. As this proof obligation cannot be discharged, the formula is not well-defined. This contrasts with SMT-LIB's approach of assuming all functions are total. As such, $1/0$ is an integer (but we don't know which one). Hence, for the SMT-LIB model in Figure 2a, Z3 returns $\mathtt{x} = 0, \mathtt{y} = 0$ as solution. Consequently, when generating random B formulae, we need to account for any WD problems as well.

We ensure well-definedness for fuzzed predicates by introducing non-discharged WD proof obligations (like $\mathtt{x} \neq 0$ above) into the formula. This can be described formally by the means of functions $\tau(\cdot)$ and $\eta(\cdot)$. Hereby, τ is our rewrite operator for B predicates whereas η is the rewrite operator for B expressions. The distinction is important as τ concatenates any WD constraints as conjuncts into the original formula whereas η must not introduce new conjuncts as this would turn the input expression into a predicate which will break typing.

Table 1: Definition of the expression rewrites by η. Unlisted expressions are not rewritten but left as-is. E, E_1, E_2 are arbitrary expressions, f is an arbitrary function, P is an arbitrary predicate, and $\odot$ is an arbitrary binary operator that is defined over expressions $(+, -, *, \div, \cup, \cap, \ldots)$.

Expression ϕ	$\eta(\phi)$
$E_1 \odot E_2$	$\eta(E_1) \odot \eta(E_2)$
$f(E)$	$f(\eta(E))$
$\mathbb{P}(S)$	$\mathbb{P}(\eta(S))$
$\{x_1, \ldots, x_n \mid P\}$	$\{x_1, \ldots, x_n \mid \tau(P)\}$
$\lambda(x_1, \ldots, x_n).\,(P \mid E)$	$\lambda(x_1, \ldots, x_n).\,(\tau(P) \wedge \mathrm{WD}(\eta(E)) \mid \eta(E))$
$\sum(x_1, \ldots, x_n).\,(P \mid E)$	$\sum(x_1, \ldots, x_n).\,(\mathrm{WD}(\eta(E)) \wedge \tau(P) \mid \eta(E))$
$\prod(x_1, \ldots, x_n).\,(P \mid E)$	$\prod(x_1, \ldots, x_n).\,(\mathrm{WD}(\eta(E)) \wedge \tau(P) \mid \eta(E))$
$\bigcap(x_1, \ldots, x_n).\,(P \mid E)$	$\bigcap(x_1, \ldots, x_n).\,(\mathrm{WD}(\eta(E)) \wedge \tau(P) \mid \eta(E))$
$\bigcup(x_1, \ldots, x_n).\,(P \mid E)$	$\bigcup(x_1, \ldots, x_n).\,(\mathrm{WD}(\eta(E)) \wedge \tau(P) \mid \eta(E))$
IF P THEN E_1 ELSE E_2 END	IF $\tau(P)$ THEN $\eta(E_1)$ ELSE $\eta(E_2)$ END

Let WD$(\cdot)$ denote the conjunction of non-discharged WD proof obligations of a predicate or B expression. We define $\eta(\cdot)$ as a function that takes a B expression and rewrites it recursively into well-defined expressions where applicable according to the rules listed in Table 1. This mostly involves expressions which have possibly non-WD predicates as operants. The final function $\tau(\cdot)$ can then be defined over the rewrite rules in Table 2. The helper function η^* hereby rewrites subexpressions within a predicate with η, for instance

$$\eta^*(\mathbb{N} = \{x \mid 1/x > 0\}) \equiv \mathbb{N} = \{x \mid x > 0 \wedge 1/x > 0\}\,.$$

For the calculation of WD$(\cdot)$, ProB's internal WD prover is used [22].

During the BanditFuzz loop, we store the raw version of the best benchmark for further mutations, i.e. the version before ensuring well-definedness, while evaluation performances on the WD-enforced predicates. Thus, mutations do not take place within the WD-ensuring parts of the input and no unnecessary WD-ensuring parts are kept (e.g. when the respective syntax node was replaced).

5 Running the Performance Fuzzer

For evaluation of how well the BanditFuzz approach works for the B method and ProB, we applied our implementation to different combinations of target and reference solvers. These experiments can be split into four categories: fuzzing only, comparing the mentioned ProB backends with another, performance fuzzing different settings for ProB's native backend, and comparing the two existing SMT-LIB translations with Z3.

Table 2: Definition of the WD-transformation τ. $P, P_1, \ldots, P_n$ are arbitrary predicates. E, E_1, E_2 are arbitrary expressions. S is an arbitrary set expression. $\Phi(\cdot)$ is an arbitrary predicate that takes an expression as argument. The function η^* rewrites every subexpression within a predicate P with η.

Formula ϕ	$\tau(\phi)$	Note
$\neg P$	$\neg\tau(P)$	
$P_1 \wedge \ldots \wedge P_n$	$\tau(P_1) \wedge \ldots \wedge \tau(P_n)$	
$P_1 \vee \ldots \vee P_n$	$\tau(P_1) \vee \ldots \vee \tau(P_n)$	
$P_1 \Rightarrow P_2$	$\tau(P_1) \Rightarrow \tau(P_2)$	
$P_1 \Leftrightarrow P_2$	$\tau(P_1) \Leftrightarrow \tau(P_2)$	
$\forall X.\,(P_1 \Rightarrow P_2)$	$\forall X.\,(\tau(P_1) \wedge \mathrm{WD}(\eta^*(P_2)) \Rightarrow \tau(P_2))$	X is a list of variables
$\exists X.\,P$	$\exists X.\,\tau(P)$	X is a list of variables
$E_1 = E_2$	$\mathrm{WD}(\eta(E_1)) \wedge \mathrm{WD}(\eta(E_2)) \wedge \eta(E_1) = \eta(E_2)$	$\neq, <, >\leq, \geq$ analogous
$E \in S$	$\mathrm{WD}(\eta(E)) \wedge \mathrm{WD}(\eta(S)) \wedge \eta(E) \in \eta(S)$	$\notin$ analogous
$E \subseteq S$	$\mathrm{WD}(\eta(E)) \wedge \mathrm{WD}(\eta(S)) \wedge \eta(E) \subseteq \eta(S)$	$\subsetneq, \not\subseteq, \not\subsetneq$ analogous

5.1 Fuzzing Only

Employing the fuzzer alone already led to the discovery of multiple issues within the respective solvers. In the case of ProB this helped improve the robustness in the case of corner-cases of well-definedness, in particular involving infinite values and formulas using the new REAL datatype which ProB now supports. Findings include:

- A series of edge cases where ProB raised an internal error and failed to produce an answer:
- Cases where ProB raised a WD error because of missing or faulty treatments in the WD analyser (in particular for certain new REAL operators such as RPOW).
- Cases where ProB raised a WD error even though the input was WD already. This led to improvements in ProB's kernel, e.g., when partitioning a predicate into components and one component is false and an earlier one has a WD error.

We did not find any cases where ProB's default solver produced a wrong answer. Additionally, we were able to consistently generate inputs which caused segmentation faults within the Z3 prover in version 4.12.2.[1] As a consequence, we changed to the Z3 nightly builds henceforth which included the respective fix.

While this highlights the benefit of input fuzzing, it does not relate to the reinforcement learning approach of BanditFuzz. However, it is noteworthy that ProB already employed a fuzzer which was not able to locate these issues before, stressing the impact of using a secondary tool chain. As we implemented a new fuzzer from scratch in this work, we were now able to generate constructs which were apparently not generated before.

[1] Reported in https://github.com/Z3Prover/z3/issues/6734.

5.2 Performance Fuzzing Between ProB's Backends

In the experiments involving backends of ProB presented in Section 2.1, we used the following settings. Firstly, the fuzz generator was set to create predicates with 2–4 conjuncts. Each of these conjuncts had a fixed AST size of 3, meaning a path from root to leaf would contain three nodes. Secondly, while ProB has begun supporting reals the support is still experimental and not fully supported by all employed backends. Thus, we did not generate any constraints involving real-valued types during the performance fuzzing (note that reals were still employed during the fuzzing-only stage of our experiments in Section 5.1).

Thirdly, we measured the Par2 score for each solver's runtime. It is defined as

$$\text{Par2}(t, r) = \begin{cases} t, & \text{if } r \in \{\text{valid}, \text{invalid}\} \\ t + \delta, & \text{if } r \in \{\text{unknown}, \text{timeout}\} \end{cases}$$

where t is the measured runtime, r is the solver's response, and δ is a fixed timeout value. In all of our experiments, we used $\delta = 2.5$ s, which is ProB's default timeout.

Another point to mention is the use of the B compiler. While not a compiler in the sense of creating machine code, the B compiler transforms a B formula by inlining any variables that it depends upon. The B compiler is essential to create so called "closures" for symbolic B values. That is, given that the variable y has the value 2, the B compiler will translate $\{x \mid x > y + 1\}$ into $\{x \mid x > 3\}$, i.e., inlining the value of y and pre-computing static subexpressions. The result is a symbolic value that can be stored and evaluated without requiring the original state (of y). The B compiler is also used to pre-compute bodies of quantifiers and while loops, and it can be used as an additional pre-processing step before calling a solver. The B compiler is invoked by both, ProB CDCL(T) and the bridge to Z3, as a preprocessing step before the solving process starts. ProB CLP(FD) does, however, not invoke the B compiler by default. This led to the discovery of examples where ProB CDCL(T) seemed to perform better than ProB CLP(FD) solely due to the B compiler being able to reduce the input to `btrue` ($\top$) or `bfalse` ($\bot$), respectively. As this does not equate to instances in which the ProB CDCL(T) solver itself performs better, we opted to incorporate the B compiler as additional preprocessing step for ProB CLP(FD) in these experiments. We will refer to the version of ProB CLP(FD) with the B compiler preprocessing as ProB[Compile].

Figure 3 shows a subset of the results produced by the performance fuzzing. Given the timeout of 2.5 s, the Par2 score should be 5 at most per benchmark. We see this does not seem to hold true in Figure 3a, where the final benchmarks had a Par2 score of around 20 for ProB CDCL(T). This looks like a bug where the timeout does not properly apply during solving, but closer investigation showed the issue to be related to the `call_cleanup/2` predicate from the SICStus Prolog standard library.

Further findings include an unhandled case for the empty set, where the internal representation was not recognised, leading to a timeout for ProB

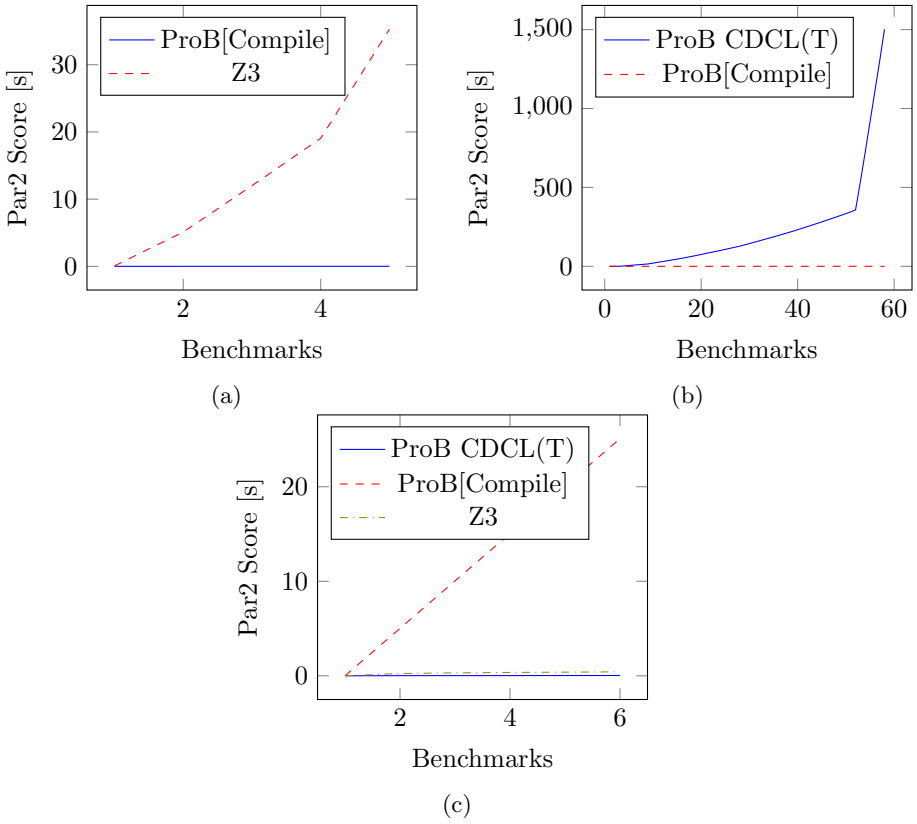

Fig. 3: Selection of performance fuzzing results between ProB[Compile], ProB CDCL(T), and Z3. All benchmarks were found within 18 h. The cactus plots show the accumulated Par2 scores up until the respective benchmarks.

CLP(FD) while ProB CDCL(T) found a solution in time. We also logged whether there were any contradictions within the solver's responses, namely if one solver reports a solution for a constraint and another saying there exists none. Such soundness bugs where found and fixed for ProB CDCL(T) and Z3. Results reported by ProB CLP(FD) were always found to be correct.

5.3 Performance Fuzzing Between ProB's Settings

While the constraint solving backends use different underlying algorithms, a comparison within one solver under different settings is testing the same algorithm, just with slightly altered behaviour in its processing steps.

We already presented ProB's default setting, ProB CLP(FD). Further settings we looked into are the addition of the CHR solver, and the SMT mode. CHR (constraint handling rules) should speed up the detection of unsolvable

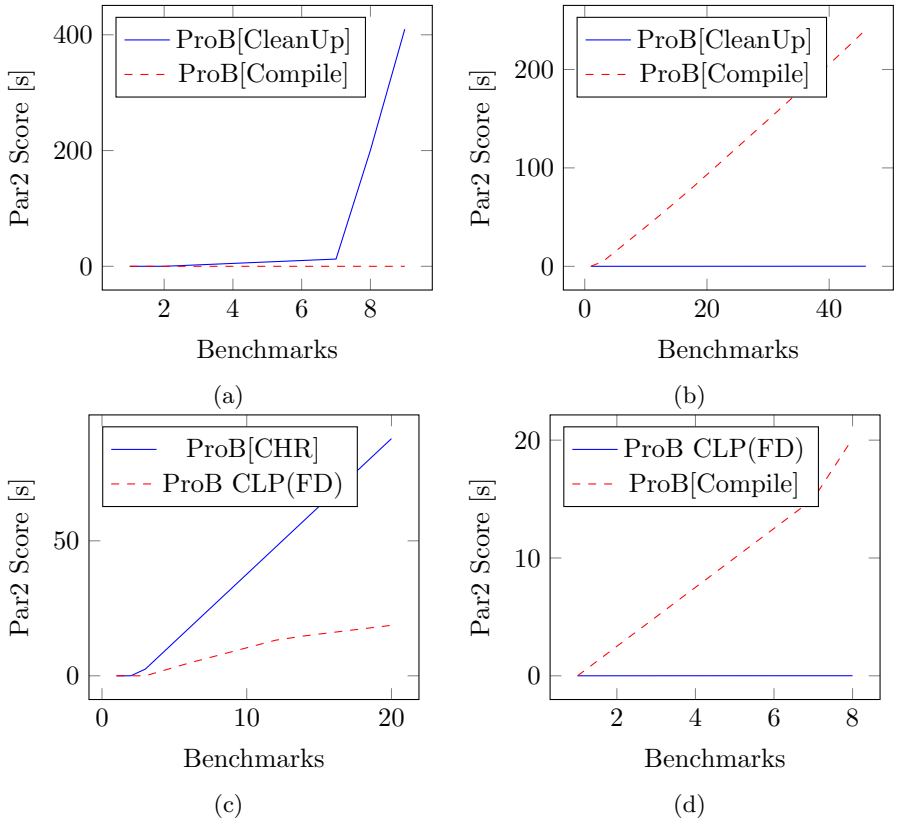

Fig. 4: Selection of performance fuzzing results between different ProB settings. The cactus plots show the accumulated Par2 scores up until the respective benchmarks.

predicates at the price of slowing down the solving process in general. The SMT mode instructs ProB to give higher priority to choice points emanating from the logical structure of the formulas, rather than relying mainly on enumeration of data values. E.g., given the predicate $x \in A \wedge (x \in B \Rightarrow P)$ the enumeration would mainly be driven by $x \in A$ in regular mode, while in SMT mode, ProB would tend to make a case distinction on $\Rightarrow$ before enumerating $x \in A$. The ProB[SMT] mode is typically more suitable for symbolic model checking tasks, while the regular mode works well with animation and explicit model checking. We will refer to both as ProB[CHR] and ProB[SMT], respectively.

Further, ProB offers the option to run a clean-up preprocessing over the input, simplifying parts of the predicate for easier solving. Activating this option should desirably not reduce performance of the solving procedure. We refer to the solver with the enabled clean-up as ProB[CleanUp], which is not to be mistaken for ProB[Compile] which involves a different preprocessing step.

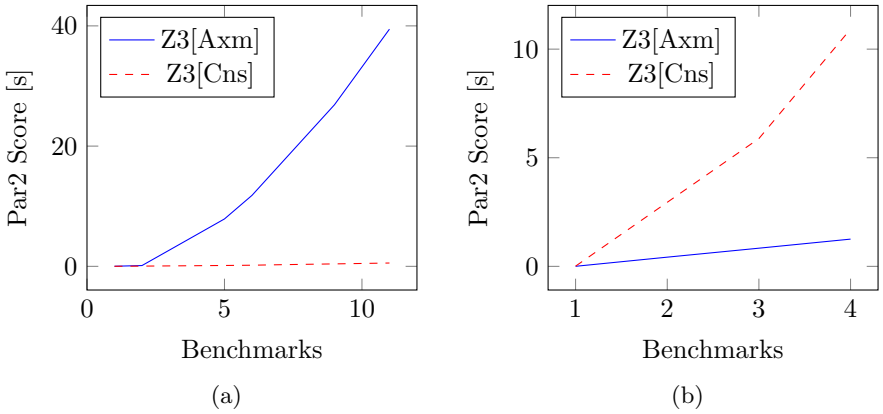

Fig. 5: Selection of performance fuzzing results between Z3[Axm] and Z3[Cns] The cactus plots show the accumulated Par2 scores up until the respective benchmarks.

Figure 4 shows the Par2 scores of a selection of the settings experiments. Note the steep increase for ProB[CleanUp] in Figure 4a from Benchmark 7 onwards where the aforementioned SICStus bug resulted in highly inflated runtimes once again. While we were capable of producing pathological inputs for almost each matchup, it remains to note that the reported benchmarks suffered from reproducibility issues. Rerunning the final benchmarks in a more isolated environment did often not mirror the expected runtime BanditFuzz reported. While this influenced to the SICStus bug, it might also indicate other clean-up problems within the solvers, as we were calling them all from within the same process, making room for shared state between them. If this state is not properly cleaned between calls, residues might accumulate, impacting subsequent performance. Further investigation is needed.

5.4 Performance Fuzzing Between the SMT-LIB translations

Finally, we applied the performance fuzzing to the two competing SMT-LIB translations described in Section 2.1. Recall that the former translation used an approach which translated the B predicates to axiomatised formulae with quantifiers, whereas the newer reimplementation reconstructs B operators by means of Z3's lambda functions. Hence, we will refer to these translations as Z3[Axm] and Z3[Cns], respectively. The solver from previous experiments in Section 5.2 simply labelled Z3 most notably made use of both translations in parallel, awaiting and propagating the faster solver's response.

Running this experiment, we quickly found a bug regarding the Z3 backend's clean-up handling during timeouts. As both, a timeout on Prolog side and a timeout within Z3, are employed, the Prolog timeout could end up interupting Z3's internal timeout clean-up, polluting internal state and increasing runtime of

subsequent calls. Figure 5 displays that after fixing this bug, we are still able to produce benchmarks in which either translation has the edge over the other.

6 Related Work

The area of performance fuzzing is comparably young but extends prior research of diagnosing performance issues and finding worst-case inputs and algorithmic complexity vulnerabilities in a static way. The first automated approaches were SlowFuzz [31] and PerfFuzz [21], both of which used mutation-based fuzzing in an evolutionary algorithm context.

Other tools quickly made use of reinforcement learning, such as Saffron [20] and PySE [18]. Saffron takes a user-specified grammar as input and refines it first with respect to found inputs which are not part of the grammar but accepted by the PUT nonetheless. The following generation of pathological inputs over the synthetic grammar then assigns probabilities to the grammar's production rules which represent their likelihood to be chosen for input generation. The final choices are then done by tournament selection until an input with maximum depth is generated. If the input increases the measured performance complexity, the corresponding production rules' probabilities are updated accordingly.

PySE [18] is a tool for automatic worst-case test generation via symbolic execution of a Python program. Here, a Q-Learning [40] approach is utilised to learn a branching heuristic to navigate through the execution paths. This heuristic is gradually updated between subsequent runs of the search for a worst-case execution.

With regard to SMT solvers, one application of performance fuzzing (next to BanditFuzz, see Section 3) is SPRFinder [42]. SPRFinder targets solver performance regressions (SPRs) for SMT string solvers by using an adapted version of BanditFuzz which targets newer releases of a string solver and uses old releases as references. Adaptions to the original BanditFuzz include the use of a set of seed inputs instead of using a single one and an adaptive configuration per loop iteration (i.e. if no SPR was found within a number of iterations, the complexity of generated inputs is subsequently increased).

7 Discussion

In this work, we implemented and adapted the BanditFuzz algorithm by Scott et al. [38] to a performance fuzzing procedure for the B method and ProB specifically. In this section, we restate the findings of our performance experiments, talk about issues we experienced during implementation of the algorithm and experiments, and iterate over lessons learned as well as take-aways for future implementation and extensions, which are worth to consider for reimplementation for other tools and formalisms as well.

Performance fuzzing results. With the help of performance fuzzing as well as simple input fuzzing, we were able to find a set of performance bugs within ProB's default and alternative solver implementations, as well as more fundamental issues within external tooling these solvers build upon. We found cases in which internal exceptions within ProB were not captured accurately, missing value interpretations regarding empty sets causing timeouts where a solution was otherwise available, soundness issues in the ProB CDCL(T) and Z3 backends, segmentation faults within the Z3 prover itself, issues with state clean-up in the Z3 backend when a timeout fires, and a performance issue within the SICStus Prolog standard library which can inflate the post-cleanup runtime after solver calls in ProB CLP(FD) and ProB CDCL(T).

Size of benchmarks. While the results make the performance fuzzing approach seem lucrative to employ, we ran into problems during implementation of the fuzz generator and reproduction capabilities of the reported results. Implementation wise, it is straight forward to create a valid B AST that ProB can technically work with, and the BanditFuzz approach promises small and concise, pathological benchmarks. Even though we restricted the size of the initially fuzzed predicates, the addition of the WD constraints in Section 4.3, inflated the constraints notably. In the future, a more involved fuzz generation which already accounts for the size of WD constraints during fuzzing might leverage the added complexity of produced benchmarks significantly. As discussed in Section 5.3, this can be caused by the found bug in SICStus Prolog regarding `call_cleanup/2` but might indicate another issue regarding subsequent calls to pathological inputs and improper clean-ups. This issue is still to be investigated, but made it harder for us to reliably produce valuable benchmarks.

Issues with snapshotting and repeated benchmarks. Another problem were imprecisions on the millisecond scale. During fuzzing for different ProB settings, we often got results where one solver performed worse in our logs but during reproduction all solvers under test exhibited the same performances. This was due to all solvers varying in runtime over a multiple hundred milliseconds, and BanditFuzz snapshotting instances where the target solver happened to perform worse — even when we already took the mean runtime of three consecutive runs for each solver. A solution might consist of employing a minimal needed improvement over the last performance margin. This would also cancel a set of produced benchmarks which had the same Par2 scores $\pm$10ms and only differed slightly while capturing the same underlying issues. This is a particular shortcoming in search for performance bugs: once one is found, due to the mutational nature of the algorithm, slightly varied inputs causing the same bug will be found as well.

Lessons learned and future perspective. We found the BanditFuzz algorithm valuable for generating pathological inputs. In our implementation, we opted to link the performance fuzzer within ProB's source code instead of making it an external tool which uses the compiled executable. Having direct access to the internals of ProB allowed us to find bugs related to post-solving clean-up

issues, such as for the Z3 backend or within SICStus Prolog. Any issues related to possibly shared state between two solver calls, i.e. subsequent queries of possibly distinct constraints, can only be caught if the program instance is the same. A draw-back is that the ProB source code is a direct dependency to the performance fuzzer. Hence, it is not possible to target different versions of the same solvers for performance regression testing as done by Zhang et al. [42]. Given that the SICStus Prolog bug we found was only reproducible for us in SICStus Prolog version 4.8.0 but not 4.7.0, the possibility of regression testing would have direct value for us as well and will be considered in the future. In retrospective, we would advise to use an external approach and keep regression testing as an easily implementable option. Implementing an API that allows the performance fuzzer to query the same solver instance subsequently should hereby still allow detection of problems with internal clean-up routines between calls.

The reported difficulty in finding reliable benchmarks targeting the different ProB settings might suggest that the fuzzed predicates were too small. For such cases, we advocate to incorporate an adaptive configuration into the algorithm, for instance as done by Zhang et al. [42]. The adaptive configuration would increase the predicate complexity until significant benchmarks are found, then should ideally aim to reduce the inflated complexity again while maintaining or increasing the performance margin.

While wallclock runtime and Par2 scores seem to be popular metrics for performance fuzzing [14, 37, 42], other performance related metrics should be accounted for as well [14]. For instance, targeting memory consumption, as done by Wen et al. [41], seems to be a valuable alternative we'd like to incorporate in future work as well.

8 Conclusions

We implemented a fuzzer for the B language which ensures well-definedness on the generated B formulae. We utilised said fuzzer within an adaption of the BanditFuzz performance fuzzing algorithm implemented within the ProB tool. Targeting different constraint solving backends and settings, our experiments successfully unveiled various performance bugs within our tooling as well as external tooling such as the Z3 prover and SICStus Prolog. Our results show that an adaption of BanditFuzz for other formalisms can become a powerful asset in debugging and fine-tuning respective tooling. The aggregated take-aways suggest benefits from implementing the performance fuzzer as an external tool which has no dependency on the version of the program under test, so as to allow regression tests over multiple releases of the program.

Acknowledgements. We want to thank our colleague Joshua Schmidt for his input and ideas regarding more targeted fuzz generation for the ProB CDCL(T) and Z3 backends. Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996). https://doi.org/10.1017/CBO9780511624162
2. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139195881
3. Abrial, J.R., Mussat, L.: On using conditional definitions in formal theories. In: ZB 2002: Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users Grenoble, France, January 23–25, 2002 Proceedings 2. pp. 242–269. Springer (Jan 2002). https://doi.org/10.1007/3-540-45648-1_13
4. Agrawal, S., Goyal, N.: Analysis of thompson sampling for the multi-armed bandit problem. In: Proceedings of the 25th Annual Conference on Learning Theory. Proceedings of Machine Learning Research, vol. 23, pp. 39.1–39.26. PMLR (Jun 2012)
5. Back, R.J.R.: On correct refinement of programs. Journal of Computer and System Sciences **23**(1), 49–68 (Aug 1981). https://doi.org/10.1016/0022-0000(81)90005-2
6. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Texts in Computer Science, Springer (2012). https://doi.org/10.1007/978-1-4612-1674-2
7. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. LNCS, vol. 13243, pp. 415–442. Springer (Mar 2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (Dec 2010)
9. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F., Voisin, L.: The first twenty-five years of industrial use of the B-method. In: Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings 25. pp. 189–209. Springer (Aug 2020). https://doi.org/10.1007/978-3-030-58298-2_8
10. Carlsson, M., Mildner, P.: SICStus Prolog—the first 25 years. Theory and Practice of Logic Programming **12**(1-2), 35–66 (Jan 2012). https://doi.org/10.1017/S1471068411000482
11. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Programming Languages: Implementations, Logics, and Programs. LNCS, vol. 1292, pp. 191–206. Springer (Sep 1997)
12. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog user's manual, vol. 3. Swedish Institute of Computer Science, Kista, Sweden (1988)
13. Chapelle, O., Li, L.: An empirical evaluation of Thompson sampling. Advances in neural information processing systems **24**, 2249–2257 (2011)
14. Chen, Y., Bradbury, M., Suri, N.: Towards effective performance fuzzing. In: 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 128–129 (Oct 2022). https://doi.org/10.1109/ISSREW55968.2022.00055

15. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14. pp. 337–340. Springer (Mar 2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Dunkelau, J., Schmidt, J., Leuschel, M.: Analysing ProB's constraint solving backends: What do they know? do they know things? let's find out! In: Rigorous State-Based Methods. LNCS, vol. 12071, pp. 107–123. Springer (May 2020). https://doi.org/10.1007/978-3-030-48077-6_8
17. Dutertre, B.: Yices 2.2. In: Computer-Aided Verification (CAV'2014). LNCS, vol. 8559, pp. 737–744. Springer (Jul 2014). https://doi.org/10.1007/978-3-319-08867-9_49
18. Koo, J., Saumya, C., Kulkarni, M., Bagchi, S.: Pyse: Automatic worst-case test generation by reinforcement learning. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 136–147 (Apr 2019). https://doi.org/10.1109/ICST.2019.00023
19. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: International Conference on Integrated Formal Methods. pp. 361–375. Springer (May 2016). https://doi.org/10.1007/978-3-319-33693-0_23
20. Le, X.B.D., Pasareanu, C., Padhye, R., Lo, D., Visser, W., Sen, K.: Saffron: Adaptive grammar-based fuzzing for worst-case analysis. SIGSOFT Softw. Eng. Notes **44**(4), 14 (Dec 2019). https://doi.org/10.1145/3364452.3364455
21. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 254–265 (Jul 2018). https://doi.org/10.1145/3213846.3213861
22. Leuschel, M.: Fast and effective well-definedness checking. In: Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings 16. pp. 63–81. Springer (Nov 2020). https://doi.org/10.1007/978-3-030-63461-2_4
23. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: The ProB constraint solver 10 years on. In: Formal Methods Applied to Complex Systems: Implementation of the B Method, chap. 14, pp. 427–446. Wiley ISTE (Jun 2014). https://doi.org/10.1002/9781119002727.ch14
24. Leuschel, M., Butler, M.: ProB: A model checker for B. In: FME. vol. 2805, pp. 855–874. Springer (Sep 2003). https://doi.org/10.1007/978-3-540-45236-2_46
25. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer **10**(2), 185–203 (Jan 2008). https://doi.org/10.1007/s10009-007-0063-9
26. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (Jun 2018). https://doi.org/10.1109/TR.2018.2834476
27. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (Nov 2021). https://doi.org/10.1109/TSE.2019.2946563
28. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS press (2009). https://doi.org/10.3233/978-1-58603-929-5-131

29. 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
30. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (May 2020)
31. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2155–2168 (Oct 2017). https://doi.org/10.1145/3133956.3134073
32. Plagge, D., Leuschel, M.: Validating B, Z and TLA+ using ProB and Kodkod. In: Formal Methods. pp. 372–386. No. 7436 in LNCS, Springer (Aug 2012). https://doi.org/10.1007/978-3-642-32759-9_31
33. Robbins, H.: Some aspects of the sequential design of experiments. Bulletin of the American Mathematical Society **55**, 527–535 (1952)
34. Schmidt, J., Krings, S., Leuschel, M.: Repair and generation of formal models using synthesis. In: Integrated Formal Methods: 14th International Conference. pp. 346–366. Springer (Sep 2018)
35. Schmidt, J., Leuschel, M.: Improving SMT solver integrations for the validation of B and Event-B models. In: Formal Methods for Industrial Critical Systems. LNCS, vol. 12863, pp. 107–125. Springer (Aug 2021). https://doi.org/10.1007/978-3-030-85248-1_7
36. Schmidt, J., Leuschel, M.: SMT solving for the validation of B and Event-B models. International Journal on Software Tools for Technology Transfer **24**, 1043–1077 (Nov 2022). https://doi.org/doi:10.1007/s10009-022-00682-y
37. Scott, J., Mora, F., Ganesh, V.: Banditfuzz: A reinforcement-learning based performance fuzzer for SMT solvers. In: Software Verification. LNCS, vol. 12549, pp. 68–86. Springer (Dec 2020). https://doi.org/10.1007/978-3-030-63618-0_5
38. Scott, J., Sudula, T., Rehman, H., Mora, F., Ganesh, V.: BanditFuzz: Fuzzing SMT solvers with multi-agent reinforcement learning. In: Formal Methods. LNCS, vol. 13047, pp. 103–121. Springer (Nov 2021). https://doi.org/10.1007/978-3-030-90870-6_6
39. Thompson, W.R.: On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika **25**(3-4), 285–294 (Dec 1933). https://doi.org/10.1093/biomet/25.3-4.285
40. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**, 279–292 (May 1992). https://doi.org/10.1007/BF00992698
41. Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., Liu, T.: Memlock: Memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 765–777. ICSE '20, Association for Computing Machinery (Jun 2020). https://doi.org/10.1145/3377811.3380396
42. Zhang, Y., Xie, X., Li, Y., Lin, Y., Chen, S., Liu, Y., Li, X.: Demystifying performance regressions in string solvers. IEEE Transactions on Software Engineering **49**(3), 947–961 (Mar 2023). https://doi.org/10.1109/TSE.2022.3168373