Synthesis of Infinite-State Systems with Random Behavior

Andreas Katis
katis001@umn.edu
University of Minnesota, USA

Grigory Fedyukovich
grigory@cs.fsu.edu
Florida State University, USA

Jeffrey Chen
chen4233@umn.edu
University of Minnesota, USA

David Greve
david.greve@collins.com
Collins Aerospace, USA

Sanjai Rayadurgam
rsanjai@umn.edu
University of Minnesota, USA

Michael W. Whalen
whalen@cs.umn.edu
University of Minnesota, USA

ABSTRACT

Diversity in the exhibited behavior of a given system is a desirable characteristic in a variety of application contexts. Synthesis of conformant implementations often proceeds by discovering witnessing Skolem functions, which are traditionally deterministic. In this paper, we present a novel Skolem extraction algorithm to enable synthesis of witnesses with random behavior and demonstrate its applicability in the context of reactive systems. The synthesized solutions are guaranteed by design to meet the given specification, while exhibiting a high degree of diversity in their responses to external stimuli. Case studies demonstrate how our proposed framework unveils a novel application of synthesis in model-based fuzz testing to generate fuzzers of competitive performance to general-purpose alternatives, as well as the practical utility of synthesized controllers in robot motion planning problems.

ACM Reference Format:
Andreas Katis, Grigory Fedyukovich, Jeffrey Chen, David Greve, Sanjai Rayadurgam, and Michael W. Whalen. 2020. Synthesis of Infinite-State Systems with Random Behavior. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416586

1 INTRODUCTION

Program synthesis aims at automated generation of implementations that meet formal specifications. It has been thoroughly explored in various contexts, such as controller synthesis and automated program repair [2, 3, 11, 14, 24, 35, 37]. The implementations are generated from of the specification's realizability and have the form of *deterministic witnesses*. Thus, by design they always compute (1) an output that meets the specification, and (2) the same output for each particular input. Determinism, however, prevents us from synthesizing systems that take advantage of *randomness* to diversify their behavior[1]. Advantages offered by these systems can be better understood when put into context of robot motion planning and fuzz testing.

Fuzz testing. Synthesis of random designs allows one to specify and create system-specific *fuzzers* [34]. The idea is to follow a mindset similar to how *model-based testing* techniques utilize the system-under-test (SUT) specification to generate test cases [45]. We propose to use the fragment of the model related to the SUT inputs to synthesize a fuzzer that repeatedly generates random (and sometimes malformed) tests. This fragment can be alternatively viewed as the fuzzer's specification, which can be further enriched with properties that dictate its behavior when certain testing objectives are met. For example, when a vulnerability is detected, we can limit the fuzzer's next generated test cases within a desired range around the test that exposed the issue. System coverage is also one such objective, where we can dictate how the fuzzer diversifies the generated tests through its specification, improving the chances of reaching previously unexplored system states. From a qualitative standpoint, synthesis in model-based fuzz testing can be considered as a viable high-level solution that does not require the user to create extensive corpora of tests. Furthermore, the synthesized fuzzers can be a strong, SUT-specific alternative to general-purpose model-based fuzzers. [4, 25, 46].

Robot motion planning. In coverage path planning problems, the goal is to maximize the area that a robot can cover while avoiding obstacles [23]. Furthermore, randomness can serve as an additional security barrier in avoidance games that involve adversaries with learning capabilities. A random strategy is inherently harder to infer and exploit. In the special case of infinite-state problems, it is an even bigger challenge, as the current state-of-the-art in automata learning is limited to finite-state problems [26].

We treat systems in the aforementioned applications as so-called *reactive systems*, which have to exhibit specification-compliant behavior against an unpredictable environment. Examples are commonly found in aviation, autonomous vehicles, and medical devices. Synthesis of random reactive designs is offered by the recent Reactive Control Improvisation (RCI) [19, 20] framework, but limited only to finite-state systems (i.e., over the boolean domain), relies on probabilistic analysis to determine the realizability of the specification, and its synthesized witnesses require further refinement to be applicable in real world scenarios.

We present a novel approach to synthesis of random infinite-state systems, whose corresponding specifications may involve constraints over the Linear Integer or Real Arithmetic theories (LIRA) [5] and thus not limited to finite-state systems. The intuition behind this effort is to allow reasoning, and consequently synthesis,

[1]For the sake of brevity, throughout the paper, we refer to such systems using the adjective *random* (e.g. *random* system/design/witness/controller).

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416586

over ranges of safe reactions instead of computing witnesses with deterministic responses.

The pursuit of generality poses new challenges, for which we propose a novel Skolemization procedure to simulate randomness. We build on top of state-of-the-art reactive synthesis approach for deterministic systems called JSyn-vg [28]. It iteratively generates a greatest fixpoint over system states that ensures the realizability of the given specification but offers only a brute and inflexible strategy for witness extraction (via predetermined Skolemization rules) [14–16]. Our key novelty is in a new algorithm that enables replacing deterministic assignments in the Skolem functions with applications of *uninterpreted random number generators*. Uninterpreted functions allow us to reason about solutions with random, broad, and most importantly, compliant behavior.

The new Skolem extraction algorithm preserves JSyn-vg's important properties. Thus, the procedure remains completely automated, unlike previous work on infinite-state synthesis that requires additional templates, or the user's intervention [3, 7, 17, 42]. More importantly, our work imposes no performance overheads over JSyn-vg, remaining thus competitive with other state-of-the-art tools which could be considered for random synthesis [35]. We implemented the Skolem extraction algorithm and applied it in two distinct case studies.

Model-based fuzz testing. We are the first to explore the applicability of reactive synthesis in fuzz testing. On a chosen set of applications designed for the DARPA Cyber Grand Challenge [18, 33], the synthesized fuzzers performed competitively against well-established tools (AFL [47], AFLFast [9]), both in terms of code coverage as well as exposing vulnerabilities.

Robot motion planning. We synthesized safe robot controllers that participate in avoidance games on both bounded and infinite arenas. Using simulation, we show how the synthesized controller leads to the robot being capable of avoiding its adversary while moving in random patterns. We demonstrate how the synthesized strategies are safe by design, no matter what bias is introduced at the implementation level. Furthermore, we showcase why randomness in the controller behavior is a mandatory feature, if synthesis is to be considered for coverage path planning problems.

To summarize, the contributions of this work are:

- the first complete formal framework that enables specification and synthesis of random infinite-state reactive systems;
- a novel Skolemization procedure that enables random synthesis with no performance overhead, by taking advantage of uninterpreted functions to reason about ranges of valid reactions;
- a novel application of synthesis in model-based fuzz testing, where we generated reactive fuzzers, yielding competitive results in terms of system coverage and vulnerability detection; and
- the application of synthesized random controllers in safety problems for robot motion planning, outlining important advantages over deterministic solutions.

The rest of the paper is structured as follows. Sect. 2 provides the necessary formal background on which our work depends. Sect. 3 illustrates and Sect. 4 describes in detail the algorithm for synthesis of random Skolem functions. The implementation is outlined in Sect. 5 and the case studies are presented in Sect 6 and Sect. 7. Finally, we discuss related work in Sect. 9 and conclude in Sect. 10.

Algorithm 1: JSyn-vg $\left(A(\vec{x},\vec{y}), G_I(\vec{y}), G_T(\vec{y},\vec{x},\vec{y}')\right)$, cf. [28].

Input: $A(\vec{x},\vec{y})$: assumptions, $G_I(\vec{y})$, $G_T(\vec{y},\vec{x},\vec{y}')$: guarantees
Output: $\langle$REALIZABLE, *Skolem*$\rangle$/UNREALIZABLE

$F(\vec{y}) \leftarrow \top$;
while $\top$ **do**
    $\phi \leftarrow \forall \vec{x}, \vec{y}.\ (F(\vec{y}) \wedge A(\vec{x},\vec{y}) \Rightarrow \exists \vec{y}'. G_T(\vec{y},\vec{x},\vec{y}') \wedge F(\vec{y}'))$;
    $\langle$*valid*, *validRegion*$(\vec{x},\vec{y})$, *Skolem*$\rangle \leftarrow$ AE-VAL(ϕ);
    if *valid* **then**
        if $\exists \vec{y}. G_I(\vec{y}) \wedge F(\vec{y})$ **then return** $\langle$REALIZABLE, *Skolem*$\rangle$;
        else return UNREALIZABLE;
    else $F(\vec{y}) \leftarrow F(\vec{y}) \wedge \neg$ExtractUnsafe(*validRegion*$(\vec{x},\vec{y})$);
end

2 BACKGROUND AND NOTATION

A first-order formula φ is satisfiable if there exists an assignment m, called a model, under which φ evaluates to $\top$ (denoted $m \models \varphi$). If every model of φ is also a model of ψ, then we write $\varphi \Rightarrow \psi$. A formula φ is called *valid* if $\top \Rightarrow \varphi$. For existentially-quantified formulas of the form $\exists y\,.\,\psi(x,y)$, validity requires that each assignment of variables in x can be *extended* to a model of $\psi(x,y)$. For a valid formula $\exists y\,.\,\psi(x,y)$, a term $sk_y(x)$ is called a *Skolem*, if $\psi(x, sk_y(x))$ is valid. More generally, for a valid formula $\exists \vec{y}\,.\,\psi(x,\vec{y})$ over a vector of existentially quantified variables $\vec{y}$, there exists a vector of individual Skolem terms, one for each variable $\vec{y}[j]$, where $0 < j \leq N$ and $N = |\vec{y}|$, such that: $\top \Rightarrow \psi(x, sk_{\vec{y}[1]}(x), \ldots, sk_{\vec{y}[N]}(x))$.

2.1 Synthesis with JSyn-vg

We build on top of JSyn-vg, a reactive synthesis procedure that takes formal specifications in the form of *Assume-Guarantee contracts*. Systems are described in terms of inputs $\vec{x}$ and outputs $\vec{y}$, using the predicate $I(\vec{y})$ to denote the set of initial outputs and $T(\vec{y},\vec{x},\vec{y}')$ for the system's transition relation, where the next (primed) outputs $\vec{y}'$ depend on the current input and state. *Assumptions* $A(\vec{x},\vec{y})$ correspond to assertions over the system's current state, while the set of *guarantees* is decomposed into constraints over the initial outputs $G_I(\vec{y})$, and guarantees $G_T(\vec{y},\vec{x},\vec{y}')$ that have to hold over any valid transition (i.e., with respect to $T(\vec{y},\vec{x},\vec{y}')$).

The algorithm behind JSyn-vg performs a realizability analysis to determine the existence of a greatest fixpoint of states meeting the contract, that can lead to an implementation. Furthermore, the computed fixpoint can be directly used for the purposes of synthesis, as it precisely captures a collection of system output constraints which, when instantiated, define safe reactions. Formally, the computed fixpoint is a set of *viable* outputs, guaranteed to preserve safety by requiring that a valid transition to another viable output is always available.

$$\mathsf{Viable}(\vec{y}) \stackrel{\text{def}}{=} \forall \vec{x}, \vec{y}.(A(\vec{x},\vec{y}) \Rightarrow \exists \vec{y}'.\ G_T(\vec{y},\vec{x},\vec{y}') \wedge \mathsf{Viable}(\vec{y}')) \quad (1)$$

The coinductive definition of viable states is sufficient to prove the realizability of a contract, as long as the corresponding decision procedure can find a viable output that satisfies the initial guarantees $G_I(\vec{y})$:

$$\exists \vec{y}. G_I(\vec{y}) \wedge \mathsf{Viable}(\vec{y})$$

Given a proof of the contract's realizability, the problem of synthesis is formally defined as the process of computing an initial output $\vec{y}_{init}$ and a function $f(\vec{x}, \vec{y})$ such that $G_I(\vec{y}_{init})$ and $\forall \vec{x}, \vec{y}. \mathsf{Viable}(\vec{y}) \Rightarrow \mathsf{Viable}(f(\vec{x}, \vec{y}))$ hold true.

Alg. 1 summarizes JSyn-vg. It begins with the generic candidate fixpoint $F(\vec{y}) = \top$ and solves the $\forall\exists$-formula ϕ for the validity (line 4) that corresponds to the definition of viable outputs in Eq. 1. If ϕ is valid and an output vector in $F(\vec{y})$ exists that satisfies the initial guarantees, then the contract is declared realizable, and a witnessing Skolem term is extracted. If ϕ is invalid, the algorithm extracts the largest subset of $F(\vec{y}) \wedge A(\vec{x}, \vec{y})$, denoted $validRegion(\vec{x}, \vec{y})$, such that the following formula is valid:

$$\forall \vec{x}, \vec{y}. \ (validRegion(\vec{x}, \vec{y}) \Rightarrow \exists \vec{y}'. G_T(\vec{y}, \vec{x}, \vec{y}') \wedge F(\vec{y}'))$$

Due to the possibility of $validRegion(\vec{x}, \vec{y})$ strengthening the assumptions $A(\vec{x}, \vec{y})$, we additionally extract a set of constraints over unsafe states (ExtractUnsafe) from $validRegion(\vec{x}, \vec{y})$. The negation of this set is then added as a new conjunct to the candidate $F(\vec{y})$ and the algorithm iterates until either ϕ is valid or $F(\vec{y}) = \bot$. For further details, we refer the reader to the original paper on JSyn-vg [28].

2.2 Realizability and Synthesis with AE-VAL

The greatest fixpoint algorithm described by JSyn-vg uses AE-VAL, an algorithm to determine the validity of $\forall\exists$-formulas and to generate (deterministic) witnesses in the form of Skolem terms. The latter feature is also the point of interest behind this work, as randomness can be introduced through replacing the part of AE-VAL's Skolemization strategy with our new proposed algorithm.

Alg. 2 gives a brief pseudocode of AE-VAL[2]. The idea is to enumerate all models of $\vec{x}$ and to extend each of them to a model of y. Because a naive enumeration would be endless, AE-VAL generates a sequence of Model-Based Projections (MBPs) [8] each of which groups models of $\vec{x}$. Formally, an MBP for model m is a formula $P(\vec{x})$, such that $m \models P(\vec{x})$, and $P \Rightarrow \exists y. \psi(\vec{x}, y)$. To create it, AE-VAL gathers all literals of ψ which are evaluated to true by m (line 6). These literals are further referred to as *Skolem constraints* π. In linear arithmetic, each Skolem constraint is composed only of arithmetic relations, linear combinations over $\vec{x}$ and y, and numeric constants. Finally, to obtain an MBP *pre*, AE-VAL just eliminates y from the conjunction of Skolem constraints (line 7).

The ExtractSk procedure, used for Skolem extraction, implements an inflexible strategy to transform Skolem constraints to local Skolem terms (we refer the reader to the original paper on AE-VAL for further details [15]). The final Skolem term has a form of a *decision tree*, where preconditions are placed on the nodes and local Skolem terms (i.e., outputs of ExtractSk) are on the leaves, i.e., the nested if-then-else structure ($ite(\cdot)$):

$$sk_y(\vec{x}) \stackrel{\text{def}}{=} ite(pre[1], sk_{1,y}(\vec{x}), ite(pre[2], sk_{2,y}(\vec{x}), \dots, (ite(pre[M-1], sk_{M-1,y}(\vec{x}), sk_{M,y}(\vec{x})))))$$

[2]Note that for simplicity of presentation, in the pseudocode we assume a single existentially quantified variable y (however, the algorithm and the implementation can handle any vector $\vec{y}$).

Algorithm 2: AE-VAL$\left(\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)\right)$, cf. [15, 16].

Input: $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$
Data: MBPs *pre*, Skolem constraints π, Skolem terms *sk*
Output: Return value $\in \{valid, invalid\}$ of $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$, *validRegion*, *Skolem*

```
1  M ← 1;
2  while ⊤ do
3      if ⋀_{i=1}^{M−1} ¬pre[i](x⃗) ⇒ ⊥ then
4          return ⟨valid, ⊤, DecisionTree(pre, sk)⟩;
5      if ∃m ⊨ ψ(x⃗, y) ∧ ⋀_{i=1}^{M−1} ¬pre[i](x⃗) then
6          π[M](x⃗, y) ← {ℓ | ℓ ∈ literals(ψ) ∧ m ⊨ ℓ};
7          pre[M](x⃗) ← QElim(y, π[M](x⃗, y));
8          sk[M](x⃗, y) ← ExtractSk(x⃗, y, ⋀_{c∈π[M](x⃗,y)} c);
9          M ← M + 1;
10     else return ⟨invalid, ⋁_{i=1}^{M−1} pre[i](x⃗), ∅⟩;
```

Finally, in the case that the input formula $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$ is invalid, AE-VAL returns $\bigvee_{i=1}^{M-1} pre[i](\vec{x})$ as the formula's maximal *region of validity*, i.e., the maximal set of models of the universally quantified variables for which the formula becomes valid. This region is used by JSyn-vg in order to further refine the candidate fixpoint during each of its iterations (Alg. 1, line 8).

3 RANDOM SYNTHESIS - MOTIVATING EXAMPLE

In this section, we demonstrate a complete run of the synthesis procedure and show how the standard synthesized witnesses are unable to exhibit random behavior. As an example, we use a safety robot motion planning problem from Neider et al. [35]. In this problem, a robot is placed on a one-dimensional grid with two players, the environment and the system, controlling its movement. Each player can choose to either move the robot left or right, or not move it at all (we refer to these choices using the values $-1, 0, 1$). The robot starts at $position = 0$, and the safety property for the system is to retain the robot in the area of the grid for which $position \geq 0$.

Fig. 1 shows an Assume-Guarantee contract for the example, described in the Lustre language. The contract has the singleton input $\vec{x} = \{x\}$ (internally identified by the `--%REALIZABLE` statement) and the outputs $\vec{y} = \{y, position\}$[3]. The contract assumption is that the environment will only make legal choices, i.e., $A(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} -1 \leq x \wedge x \leq 1$. The initial guarantee refers to the initial position of the robot and the system choices for movement, i.e., $G_I(\vec{y}) \stackrel{\text{def}}{=} (position = 0) \wedge (-1 \leq y \wedge y \leq 1)$. On the other hand, the transitional guarantee captures the safety property along with the stateful computation step for the new position, i.e., $G_T(\vec{y}, \vec{x}, \vec{y}') \stackrel{\text{def}}{=} (position' = position + x + y') \wedge (position' \geq 0) \wedge (-1 \leq y' \wedge y' \leq 1)$ (the transition relation for *position* is defined using Lustre's `->` and

[3]Variable *property* is local to the contract. Formally, local variables are treated as system outputs.

```
node onedim(x,y : int) returns();
var
  ok1, ok2 : bool;
  position : int;
let
  assert x >= -1 and x <= 1;
  position = 0 -> (pre(position) + x + y);
  ok1 = y >= -1 and y <= 1;
  ok2 = position >= 0;

  --%PROPERTY ok1;
  --%PROPERTY ok2;
  --%REALIZABLE x;
tel;
```

Figure 1: Assume-Guarantee contract in Lustre.

```
void skolem() {
  if(position + x == 1) {
    y = -1;
  } else if (position + x >= -1 &&
        position + x <= 0) {
    y = - (position + x);
  } else {
    y = 0;
  }
}
```

Figure 2: Synthesized deterministic witness in C.

`pre` operators).[4] The safety properties are captured by `ok1` and `ok2` (declared as such using `--%PROPERTY`).

The procedure begins with a call to JSyn-vg using the contract as its input. The contract is realizable and the greatest fixpoint of viable states is $F(\vec{y}) \stackrel{\text{def}}{=} property \geq 0$. AE-VAL declares that the formula $\phi \stackrel{\text{def}}{=} \forall \vec{y}, \vec{x}.(A(\vec{x}, \vec{y}) \wedge F(\vec{y}) \Rightarrow \exists \vec{y}'.G_T(\vec{y}, \vec{x}, \vec{y}') \wedge F(\vec{y}')$ is valid and extracts a Skolem term as a witness. Fig. 2 presents a direct translation of the function to C. The synthesized implementation behaves in a deterministic way under the following conditions:

(1) whenever $position + x = 1$, the system chooses to move left ($y = -1$);
(2) if $position + x$ equals 0 or -1, then the system chooses to do nothing or move right, respectively ($y = -(position + x)$);
(3) for any other case, the system chooses to do nothing ($y = 0$).

While the implementation preserves safety, the set of possible actions are limited due to the deterministic assignments to the output y. Interestingly, for this particular implementation the system forces the robot to go back to positions that are dangerously close to the unsafe region! Similarly, the corresponding solution by Neider et al. is the winning set $[0, 3)$, which would translate to implementations where the system would never move the robot beyond $position = 2$. Nevertheless, implementations exist for which the system can exercise a broader set of behaviors. For this example in particular, when either condition (1) or (3) is true, the system can freely choose any possible move action without violating the safety properties. Fig. 3 shows an implementation that can (theoretically) exercise any such possible assignment (we explain why in Sect. 4.2). In the following sections, we present a new method to synthesize a random witness that can, in theory, provide all such possible permutations using a single implementation.

[4] Intuitively, the problem is formalized in a manner where the position of the robot is updated after both players make a choice, with the system reacting to the choice of the environment.

4 SYNTHESIZING RANDOM DESIGNS

The standard Skolem term extraction algorithm in AE-VAL does not support the generation of Skolem functions with random variable assignments. In this section, we present a new procedure to compute witnesses that can be used to simulate random behavior.

4.1 Overview

Our proposed algorithm preserves the overall structure of AE-VAL as well as the soundness of its results [15]. The main idea is to replace the deterministic assignments that eventually appear in the leaves of the generated decision tree with applications of uninterpreted functions, which when translated at the implementation level, can be viewed as function calls to a user-defined random number generator. We refer to these functions as *uninterpreted random number generators*:

Definition 4.1 (Uninterpreted Random Number Generator (URNG)). URNG is an uninterpreted function

$$f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, l, u) : T_1 \times \ldots \times T_{|D|} \times \mathbb{B} \times \mathbb{B} \times T \times T \rightarrow T,$$

where $T : \{\mathbb{Z}, \mathbb{R}\}$, H is a collection of right side expressions extracted from the set of disequalities D, ℓ and u determine the bounded interval for the randomly generated value, and $\ell_{closed}, u_{closed}$ are boolean flags that, when set, identify the corresponding bound as being closed. Without loss of generality, we require the following postconditions to hold, on any supplied implementation of f_{RNG}:

(1) $\forall h \in H. f_{\text{RNG}}(H, \_, \_, \_, \_) \neq h$

Algorithm 3: ExtractSk$(\vec{x}, y, \pi)$

Input: Variables $\vec{x}$, y, Skolem constraints
$$\pi(\vec{x}, y) = \bigwedge_{r \in E \cup D \cup G \cup GE \cup L \cup LE} r(\vec{x}, y)$$
Output: Term sk, such that $(y = sk(\vec{x})) \Rightarrow \pi(\vec{x}, y)$

$\ell_{closed} \leftarrow \bot$, $u_{closed} \leftarrow \bot$;
if $E \neq \varnothing$ **then return** ASN(e), s.t. $e \in E$;
if $G \cup GE \neq \varnothing$ **then**
  $\ell \leftarrow$ MAX$(G \cup GE)$;
  $\ell_{closed} \leftarrow G = \varnothing \vee$ MAX$(G) <$ MAX(GE);
if $L \cup LE \neq \varnothing$ **then**
  $u \leftarrow$ MIN$(L \cup LE)$;
  $u_{closed} \leftarrow L = \varnothing \vee$ MIN$(L) >$ MIN(LE);
if $\ell(\vec{x}) = u(\vec{x})$ **then return** ℓ;
$H \leftarrow \{$ASN$(d) \mid d \in D\}$;
if $\ell =$ undef $\wedge$ $u =$ undef **then return** $f_{\text{RNG}}(H, \top, \top, -\infty, +\infty)$;
if $\ell =$ undef **then return** $f_{\text{RNG}}(H, \top, u_{closed}, -\infty, u)$;
if $u =$ undef **then return** $f_{\text{RNG}}(H, \ell_{closed}, \top, \ell, +\infty)$;
return $f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, \ell, u)$

(2) $f_{\text{RNG}}(H, \bot, \bot, \ell, u) \in (\ell, u)$
(3) $f_{\text{RNG}}(H, \bot, \top, \ell, u) \in (\ell, u]$
(4) $f_{\text{RNG}}(H, \top, \bot, \ell, u) \in [\ell, u)$
(5) $f_{\text{RNG}}(H, \top, \top, \ell, u) \in [\ell, u]$

The use of URNGs allows us to reason about valid regions of values for variable assignments instead of a particular value. Furthermore, the postconditions defined for these functions play an integral role in determining the soundness of the resulting Skolem function. It is important to note that we do not have to reason regarding the emptiness of the intervals. The intuition behind this is that such computed constraints infer an unrealizable contract. In these scenarios AE-VAL would declare the input ∀∃-formula as invalid, and the Skolem extraction algorithm would never be invoked.

4.2 Algorithm

Alg. 3 shows our proposed procedure for extracting Skolem functions that allow for random choices. It is invoked from Alg. 2 and takes a set of universally quantified variables $\vec{x}$, an existentially quantified variable y, and Skolem constraints π computed in Alg. 2. Alg. 3 constructs a graph of a function that is embedded in a relation, specified by a conjunction of expressions over the relational operators $\{=, \neq, >, \geq, \leq, <\}$, using the following constraints:

$$E \stackrel{\text{def}}{=} \{y = f_i(x)\}_i \quad D \stackrel{\text{def}}{=} \{y \neq f_i(x)\}_i \quad G \stackrel{\text{def}}{=} \{y > f_i(x)\}_i$$
$$GE \stackrel{\text{def}}{=} \{y \geq f_i(x)\}_i \quad LE \stackrel{\text{def}}{=} \{y \leq f_i(x)\}_i \quad L \stackrel{\text{def}}{=} \{y < f_i(x)\}_i$$

In addition to the constraints above, Alg. 3 also utilizes the following helper functions (where $\sim \in \{<, \leq, =, \neq, \geq, >\}$):

$$\text{ASN}(y \sim e(x)) \stackrel{\text{def}}{=} e \quad \text{MIN}(\{s\}) \stackrel{\text{def}}{=} \text{ASN}(s) \quad \text{MAX}(\{s\}) \stackrel{\text{def}}{=} \text{ASN}(s)$$
$$\text{MIN}(S) \stackrel{\text{def}}{=} ite(\text{ASN}(s) \,{}_{(}\!\leq_{)} \text{MIN}(S \setminus \{s\}), \text{ASN}(s), \text{MIN}(S \setminus \{s\})), s \in S$$
$$\text{MAX}(S) \stackrel{\text{def}}{=} ite(\text{ASN}(s) \,{}_{(}\!\geq_{)} \text{MAX}(S \setminus \{s\}), \text{ASN}(s), \text{MAX}(S \setminus \{s\})), s \in S$$

Operator MIN (MAX) computes a symbolic minimum (maximum) of the given set of constraints. While the algorithm is applicable for both LIA and LRA, the following transformations are used for the case of integers:

$$\frac{A < B}{A \leq B - 1} \qquad \frac{A \geq B}{A > B - 1}$$

These transformations help avoid clauses containing $<$ and $\geq$. Line 1 initializes the value of the boolean flags ℓ_{closed} and u_{closed} to false, and line 2 handles the case where equality constraints exist over y. Lines 3 to 8 construct the expressions for the lower and upper bounds, and the truth of the flags depends on the (symbolic) comparison between the symbolic minima and maxima. Line 9 handles the case where the lower bound is equal to the upper bound. It should be noted that for cases handled by lines 2 and 9 only deterministic choices exists.

Lines 10 to 14 attempt to compute an expression containing a URNG that considers the set of disequalities D. First, the algorithm extracts the right-hand side of disequalities in line 10. If both bounds are undefined, line 11 returns the application of the URNG $f_{\text{RNG}}(H, \top, \top, -\infty, +\infty)$, where $-\infty$ and $+\infty$ are represented as free variables that can be later mapped respectively to the minimum and maximum arithmetic representations supported by the implementation (e.g. `INT_MIN` and `INT_MAX` for integers in C). If only the lower bound is undefined (line 12), we use $f_{\text{RNG}}(H, \top, u_{closed}, -\infty, u)$ to generate a random value with an unconstrained lower bound. Similarly, we handle the case where no constraints exist for the upper bound in line 13. In line 14, both ℓ and u are defined and the algorithm returns $f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, \ell, u)$ to capture a random value within the respective bounds. In all above cases, when $H \neq \emptyset$, the URNG is expected to generate a value that satisfies all disequality constraints in D. For the special case where $D = \emptyset$, there are no such disequalities over y and the Skolem term can freely assign any value within the computed bounds ℓ and u.

As an illustration of our procedure, we present summarized runs over the following examples.

Example 4.2. Consider the formula $\forall x.\exists y_1, y_2.\psi(x, y_1, y_2)$ over LIA, where:

$$\psi(x, y_1, y_2) \stackrel{\text{def}}{=}$$
$$(x \leq 2 \wedge y_1 > -3x \wedge y_2 < x) \vee (x \geq -1 \wedge y_1 < 5x \wedge y_2 > x)$$

The formula is valid since there exists an assignment to y_1 and y_2 that satisfies the constraints in ψ, for any x. In order to construct such a witness, AE-VAL needs to consider two separate cases for x, i.e., the constraints $x \leq 2$ and $x \geq -1$.

Under $x \leq 2$, the deterministic Skolem terms would be $-3x + 1$ for y_1 and $x - 1$ for y_2. For the random case, Alg. 3 computes $f_{\text{RNG}, y_1}(\emptyset, \bot, \top, -3x, +\infty)$ and $f_{\text{RNG}, y_2}(\emptyset, \top, \bot, -\infty, x)$. Under $x \geq -1$, the deterministic terms would be $-3x+1$ for y_1 and $x+1$ for y_2, while Alg. 3 computes the functions $f_{\text{RNG}, y_1}(\emptyset, \top, \bot, -\infty, 5x)$ and $f_{\text{RNG}, y_2}(\emptyset, \bot, \top, x, +\infty)$, respectively.

Note that for the random case, the above terms are finally combined into the Skolem term for $\exists y_1, y_2.\psi(x, y_1, y_2)$:

$$sk_{\vec{y}}(x) \stackrel{\text{def}}{=} ite(x \leq 2,$$
$$(y_1 = f_{\text{RNG}, y_1}(\emptyset, \bot, \top, -3x, +\infty) \wedge y_2 = f_{\text{RNG}, y_2}(\emptyset, \top, \bot, -\infty, x)),$$
$$(y_1 = f_{\text{RNG}, y_1}(\emptyset, \top, \bot, -\infty, 5x) \wedge y_2 = f_{\text{RNG}, y_2}(\emptyset, \bot, \top, x, +\infty)))$$

Example 4.3. Consider an Assume-Guarantee contract for a system with the input vector $\vec{x} = \{x_1, x_2, x_3\} \in \mathbb{R}^3$ and one output $y \in \mathbb{R}$ and the following constraints

- $A(x_1, x_2) \stackrel{\text{def}}{=} x_1, x_2 \in (0, 1)$
- $G_I(y) \stackrel{\text{def}}{=} \top$
- $G_T(y, x_1, x_2, y') \stackrel{\text{def}}{=} y' \in (0, 1) \wedge y' \neq x_1 \wedge y' \neq x_2$

The above specification is realizable as there are infinitely many assignments to y that satisfy the guarantees G given any value of x_1, x_2 in $(0, 1)$. Using Alg. 3 we retrieve the following Skolem term to enable random behavior (note that input x_3 is not included in the set H, i.e. the first argument of the function):

```
void skolem() {
  if(position + x == 1) {
    y = RandVal(1, 1, -1, 1);
  } else if (position + x >= -1 &&
        position + x <= 0) {
    y = - (position + x);
  } else {
    y = RandVal(1, 1, -1, 1);
  }
}
```

Figure 3: Synthesized random witness.

```
double RandVal(_Bool lflag, _Bool uflag,
  int min = lflag ? lbound : lbound+1;
  int max = uflag ? ubound : ubound-1;
  int range = max - min + 1;
  double rnd = (double) rand() /
        (1.0 + (double) RAND_MAX);
  int value = (int) ((double) range * rnd);
  return value + min;
}
```

Figure 4: Example random number generator.

$$sk_y(\vec{x}) \stackrel{\text{def}}{=} f_{\text{RNG},y}(\{x_1, x_2\}, \bot, \bot, 0, 1).$$

Example 4.4. Consider the contract from Fig. 1. The details of the synthesis procedure remain identical with the deterministic approach up until the Skolemization step. Fig. 3 shows the C implementation for the random witness that is synthesized using Alg. 3. Our proposed Skolemization procedure returns the assignment value for y that is equivalent to $f_{\text{RNG}}(\top, \top, -1, 1)$, for the conditions under which the system can safely choose to move the robot either left, right, or not at all. The actual choice is randomly made through the application of a function named *RandVal*. The implementation of the function is then left to the engineer's discretion (an example is given in Fig. 4).

It is important to note that throughout this section, we presented Alg. 3 assuming that the input formula $\psi(\vec{x}, y)$ to AE-VAL is *disjunction-free*. The main difference between the original Skolemization procedure in AE-VAL and our new one is that the former does not provide all possible Skolem terms, while the latter does. Thus, the original AE-VAL algorithm is not sensitive to the shape of the original formula, but our new algorithm requires a special treatment for the disjunctions. In fact, our approach is generalizable to the case of arbitrary formulas via converting them the Disjunctive Normal Form (DNF). The set of Skolem functions can then be generated for each of the disjunct separately. The final solution is then composed from these *partial* Skolem functions along with a parameter that randomly picks one of them.

4.3 Soundness and Completeness

In this section we prove that Alg. 3 is sound and can provide all possible Skolem terms given a set of Skolem constraints $\pi(\vec{x}, y)$. As we noted in the beginning of Sect. 4.2, Skolem constraints are created from literals of a formula in linear arithmetic, thus it cannot contain disjunctions.

Theorem 4.5 (Soundness of Skolem Extraction). *Assuming that the properties 1-5 from Def. 4.1 hold, Alg. 3 returns valid Skolem terms.*

Proof. To prove this statement, it suffices to show that any computed Skolem term $sk_y(\vec{x})$ by Alg. 3 accompanied by the associated postconditions in Def. 4.1, implies the input Skolem constraints in $\pi(\vec{x}, y)$. Return lines 2 and 9 in ExtractSK are trivial cases, as they reduce to a simple assignment from equality constraints. Line 11 refers to the case where no bounds have been defined and the computed Skolem term is a URNG that utilizes the unconstrained variables $-\infty$ and $+\infty$ along with postcondition 5 to ensure the choice of an arbitrary value within the specified domain. Lines 12 to 13 handle the case where inequalities exist that determine the lower and upper bounds $\ell(\vec{x})$ and $u(\vec{x})$. If the lower bound is undefined, line 12 returns a URNG that is guaranteed to provide a random value between $-\infty$ and u as per postconditions 4 and 5. We prove the soundness of terms provided by line 13 in a similar manner. If both bounds exist, then in line 14 the Skolem term returned is a URNG guaranteed to provide a value within the range specified by $\ell(\vec{x}), u(\vec{x})$, as per postconditions 2-5. □

Theorem 4.6 (Completeness of Skolem Extraction). *The Skolem terms generated by Alg. 3 are sufficient to represent all possible witnesses of the conjunctive ∀∃-formula in Eq. 1.*

Proof. It suffices to prove that no weaker set of postconditions pc' (i.e., $pc \Rightarrow pc'$) exists, such that:

$$\forall \vec{x}.pc'(sk(\vec{x})) \Rightarrow \pi(\vec{x}, sk(\vec{x})) \tag{2}$$

We prove this by contradiction, assuming that pc' exists whenever Alg. 3 returns.

Lines 2 and 9. Alg. 3 returns the deterministic assignments ASN(e) and ℓ, for which no weaker postconditions exist.

Line 11. In this case, no bounds have been defined, and postcondition 5 is used to denote a range with unconstrained bounds $-\infty$ and $+\infty$. Formally, we can simplify this postcondition to $pc = true$, for which no weaker postcondition exists. It is also noteworthy to state that weaker postconditions exist, but all of them have to violate at least one disequality in D.

Line 12. We have ℓ = undef, i.e., no constraints exist for the lower bound, and the Skolem term

$$sk(\vec{x}) = f_{\text{RNG}}(H, \top, u_{closed}, min, u)$$

is returned. Depending on whether the upper bound u is closed or not, we have two cases. For brevity, we show the proof for the case where u is closed, and the corresponding case for the open bound follows similar principles.

- When u is closed, the output constraints are simplified to $\pi(\vec{x}, sk(\vec{x})) = sk(\vec{x}) \leq u$ and the Skolem term

$$sk(\vec{x}) = f_{\text{RNG}}(H, \top, \top, -\infty, u)$$

is returned, with postcondition 5 capturing the term's range. Assume that a weaker postcondition pc' exists, such that Eq. 2 holds. Without loss of generality, we pick

$$pc' = f_{\text{RNG}}(H, \top, \top, -\infty, u) \in [-\infty, u']$$

with $u' > u$. Therefore, we have that $pc \Rightarrow pc'$, but Eq. 2 does not hold for pc', as the new term may provide the value u' as an output, falsifying $\pi(\vec{x}, sk(\vec{x}))$.

Line 13. Similar to proof for line 12.

Line 14. $\ell \neq u \neq$ undef, and as such the output constraints can be simplified into $\pi(\vec{x}, sk(\vec{x})) = \ell \sim sk(\vec{x}) \sim u$, where $\sim\ \in \{<, \leq\}$. We have the following cases corresponding to the possible ranges:

1. (ℓ, u). In this case we have $sk(\vec{x}) = f_{\text{RNG}}(H, \perp, \perp, \ell, u)$ and as postcondition pc the second postcondition from Def 4.1. Assume that a weaker postcondition pc' exists, such that Eq. 2 holds. We can pick $pc' = f_{\text{RNG}}(H, \perp, \perp, \ell, u) \in [\ell, u]$. In this case, $pc \Rightarrow pc'$ holds, but Eq. 2 does not hold, as we can pick any of the assignments $sk(\vec{x}) = \ell$, $sk(\vec{x}) = u$, which violate the constraints in π, reaching a contradiction.
2. $[\ell, u)$. Similar to the previous proof, by picking, e.g., $pc' = f_{\text{RNG}} \in [\ell, u]$.
3. $(\ell, u]$. Similarly, we can pick $pc' = f_{\text{RNG}} \in [\ell, u]$.
4. $[\ell, u]$. Similarly, we can pick $pc' = f_{\text{RNG}} \in [\ell', u]$, where $\ell' < \ell$.

□

5 IMPLEMENTATION AND EVALUATION

We implemented our random synthesis algorithm as a complementary procedure to the original synthesis framework JKind [21], a Java implementation of a popular Kind model checker [12, 21, 28]. Following Kind, the input contracts are expressed using the Lustre dataflow language [27]. JKind provides support for synthesis both through the fixpoint algorithm in JSyn-vg as well as its predecessor, JSyn, a realizability checking algorithm based on the k-induction principle [22, 29, 30]. Our proposed Skolemization procedure in Alg. 3 is a new extraction method that is performed after the validity checking procedure in AE-VAL, thus making it inherently compatible with both JSyn and JSyn-vg [5]. It is noteworthy that our approach does not add any performance overhead to the baseline implementation of JSyn-vg, as shown in Table 2.

Since the synthesized Skolem functions are expressed in the SMT-LIB 2.0 language [5] by default, we translate them into executable C implementations. For the purposes of this paper, we mapped the application of URNGs to calls to random number generators of uniformly distributed values, unless otherwise noted.

The evaluation process of our work is twofold:

1. *Empirical*. We performed case studies in applications where synthesis of random designs can be beneficial.[6] For the first case study, we conducted an experiment in the context of model-based fuzz testing, where the goal was to synthesize *reactive graybox fuzzers* capable of exposing vulnerabilities that can crash an application, through random test case generation. The second study revolves around *controller synthesis* for avoidance games in robot motion planning.
2. *Synthesis time*. We investigated the effect that our Skolemization algorithm had on JSyn-vg in terms of synthesis time. Furthermore, we compared our work to DT-Synth, a state-of-the-art synthesis tool for infinite-state problems [35].

[5]The modified version of JSyn-vg for random synthesis is available at https://github.com/andrewkatis/jkind-1/tree/synthesis. The modified version of AE-VAL with the new Skolemization procedure is available at https://github.com/andrewkatis/fuzzersynthesis.

[6]The benchmarks are publicly accessible online. DOI : 10.6084/m9.figshare.12228026

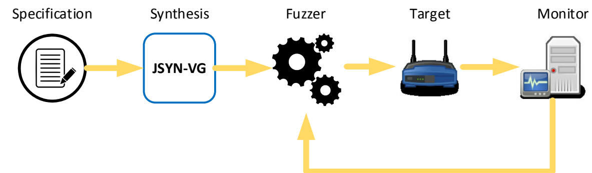


Figure 5: Fuzzer synthesis and testing diagram

6 CASE STUDY 1: REACTIVE FUZZERS

In our first case study, we explored the applicability of synthesized implementations with random behavior in fuzz testing. We focused on model-based approaches to examine a system-under-test (SUT), the input specification of which was used to derive test cases (see Utting et al. for a detailed survey [45]). In the past, model-based fuzz testing revolved around the use of structured descriptions of the system input in the form of grammars and a sophisticated implementation of a fuzzer that, given a grammar, would continuously feed random inputs to the SUT [1, 4, 39, 46]. We show that synthesis offers a viable alternative technique in this context, where the generated implementations can serve as SUT-specific fuzzers, requiring for configuration nothing but the input specification for the SUT.

6.1 Setup and Evaluation

The main intuition is that the SUT's input description can be viewed as a substantial fragment of the fuzzer's specification, which can then be used to synthesize a reactive random test case generator. Fig. 5 depicts our exact setup, where the designer already has a specification for the SUT and uses JSyn-vg with our Skolemization algorithm to automatically generate a corresponding fuzzer. The fuzzer is then attached to the SUT (Target), along with an accompanied monitoring service (Monitor) that tracks progress with respect to the SUT-related statistics (e.g., coverage). Following the definition of *graybox fuzzing*, a feedback loop exists where monitored information can be subsequently fed to the fuzzer, in order to dictate the generation of future test cases.

Using this setup, we proceeded with a thorough performance evaluation of our synthesized fuzzers, following guidelines that were recently proposed by Klees et al. [31]:

SUT Selection. We considered ten applications from the DARPA Cyber Grand Challenge (CGC) [7], a benchmark collection that has been extensively used in the past to assess the performance of fuzzers due to the high degree of interactivity between the SUT and the user [38, 41, 44]. The original collection was aimed towards the evaluation of automated reasoning and testing tools, and each application is intentionally documented in a way that is insufficient

[7]The public CGC benchmark collection is available at https://bit.ly/2HBqrJq.

Table 1: Fuzzer performance comparison and synthesis times.

	AFL				AFLFast				Synthesized Fuzzer		
	w/o corpus		w/ corpus		w/o corpus		w/ corpus				
System Under Test	**Coverage (%)**	**Crashed?**	**Coverage (%)**	**Crashed?**	**Coverage (%)**	**Crashed?**	**Coverage (%)**	**Crashed?**	**Coverage (%)**	**Crashed?**	**Synthesis times (s)**
basic_messaging	**83.76%**	✘	82.48%	✘	82.48%	✘	82.48%	✘	81.21%	✔	16.067
Dive_Logger	92.18%	✘	92.41%	✘	92.18%	✘	92.18%	✘	**93.79%**	✔	99.852
Divelogger2	80.50%	✘	**87.04%**	✘	78.40%	✘	85.08%	✘	83.25%	✘	77.689
Email_System_2	78.71%	✘	**92.90%**	✘	91.61%	✘	**92.90%**	✘	84.84%	✔	36.624
Movie_Rental_Service	38.32%	✘	38.72%	✘	38.32%	✘	38.72%	✘	**49.50%**	✔	140.826
Palindrome	**78.13%**	✔	**78.13%**	✔	**78.13%**	✔	**78.13%**	✔	75.00%	✔	1.231
PTaaS	71.94%	✔	**78.06%**	✔	46.76%	✔	46.76%	✔	74.46%	✔	77.041
Quadtree_Conways	67.04%	✘	**84.08%**	✘	67.04%	✘	67.04%	✘	64.79%	✘	88.743
SCUBA_Dive_Logging	80.97%	✔	80.67%	✔	76.41%	✔	76.41%	✔	**83.56%**	✔	101.813
User_Manager	58.43%	✘	67.45%	✘	29.02%	✘	29.02%	✘	**79.80%**	✘	16.289

to derive a precise specification from the documents themselves. To simulate the context under which synthesis would make most sense as a tool, we closely inspected and ran each application in order to identify the types and sequences of inputs each application takes. The manual process of discovering and writing input specifications per application was non-trivial, as each application differs considerably from the rest, and was the main factor to the study being limited to a subset of ten applications.

Fuzzer specification. After inspecting each application to identify the kinds of inputs that it takes, we wrote a corresponding Assume-Guarantee contract for a fuzzer. Each fuzzer specification consists of properties that capture the valid ranges of values for each one of the SUT inputs. Moreover, the specification is stateful, making each fuzzer reactive to changes (or lack thereof) in coverage results from previously generated tests. We specified the behavior of the fuzzer in such a way that, for the majority of its runtime, valid inputs are fed into the SUT. As long as no progress is made in terms of coverage, the fuzzer attempts generating invalid tests with probability $p = 0.2$.

Formalization. All of the aforementioned elements that comprise the fuzzer specification can be expressed using a set of *safety* properties over the SUT inputs, where each set precisely captures the conditions under which a (in)valid value is generated for the corresponding input. An example pair of such properties is the following:

- $prop_1 \stackrel{\text{def}}{=} p' \geq 0 \land p' \leq 1$
- $prop_2 \stackrel{\text{def}}{=} (\neg cvg \land (p \leq 0.1 \lor p \geq 0.9)) \Rightarrow in'_{\text{sys}} \notin S_{\text{valid}}$

Variables p and in_{sys} are fuzzer outputs, with in_{sys} also serving as a corresponding input for the SUT. The value of $p \in [0, 1]$ is picked randomly for each test, and it determines whether the next (primed) system input in'_{sys} will be assigned to a valid value (i.e., a value in S_{valid}) or not. Variable cvg is an input to the fuzzer and can be viewed as a flag which, when set, informs the fuzzer that the previous test resulted in progress in system coverage (e.g., line coverage improved). If such progress was not observed, then we allow the fuzzer to randomly consider invalid values in subsequent tests. More specifically, when $p \leq 0.1 \lor p \geq 0.9$, the fuzzer will generate an invalid value, i.e., a value that does not satisfy the constraints that define S_{valid}. Following the notation that we described in previous sections, the synthesis problem for the properties above is to ensure that $\forall p, cvg, in_{\text{sys}} \exists p', in'_{\text{sys}}.(prop_1 \land prop_2)$ is valid.

Synthesis and Evaluation. Using the fuzzer contracts, we synthesized a fuzzer for each application and ran it against the SUT using the setup in Fig. 5. We set the timeout for each fuzzing campaign to nine hours, and monitored the SUT line coverage (*gcov*) as well as crashes. To compare performance, we also ran fuzzing campaigns using AFL [47] and AFLFast [9], using their default configurations. We selected these tools primarily due to AFL being one of the most prominent tools in the area, while AFLFast is a recent extension to AFL that has been shown to perform better with respect to vulnerability detection.[8] Both tools were run both with and without an initial corpus in order to provide a more complete picture of their performance, whether the user provides additional information or not. To remain fair with respect to the evaluation, the corpora were created using tests that exercise application locations that are as deep as possible.

Table 1 shows the results of our experiments. Most of the applications contain unreachable code related to debugging methods, and as such 100% coverage is not attainable using *gcov* without further modifications to the source code. While we were able to achieve ≥ 75% line coverage for the majority of the benchmarks, the application "Movie_Rental_Service" was the worst performing with only 49.5%. Despite that, the synthesized fuzzer outperformed both AFL and AFLFast on either configuration with a significant margin. In fact, our synthesized fuzzers outperformed both AFL and AFLFast on four applications and remained within 4% of the best performing tool for five others, with "Quadtree_Conways" being the only exception. More interestingly, seven of the synthesized fuzzers were able to crash the corresponding application at least once, whereas AFL/AFLFast were only able to crash three.

Considering the performance results along with the low synthesis time per fuzzer, we believe that synthesis of model-based fuzzers should be considered a viable tactic towards testing systems where a specification already exists. Arguably, a synthesized fuzzer is as easy to use as a general-purpose tool like AFL. Furthermore, the user does not have to provide additional information through a corpus, a procedure in testing that often times can be time consuming and cumbersome, as both valid and invalid input sequences have to be considered for a successful campaign.

7 CASE STUDY 2: ROBOT MOTION PLANNING

In our second study, we synthesized implementations for robots participating in two-player safety games against an adversary. The study is furthermore split into two parts.

[8] Both AFL and AFLFast do not support line coverage reporting natively. To monitor coverage, we used *afl-cov* [40], a wrapper tool that enables the use of *gcov* with AFL and its variants.

7.1 Simulating avoidance games

We experimented on simulating an avoidance game in a bounded arena, where the synthesized solution was used against two different adversarial scenarios. Both the properties of the robot and the adversary were specified using their position in terms of (x, y) coordinates. Formally, we described the game using the following properties:

- Initial state : The robot starts in $(x_{\text{init}}, y_{\text{init}})$ (similarly for the adversary).
- Valid transitions : $x'_{\text{robot}} \in [x_{\text{robot}} - \delta, x_{\text{robot}} + \delta]$, where δ is user-defined and captures the maximum distance between subsequent moves (similarly for y-coordinate and the adversarial transitions).
- In-bounds property : $x_{\text{robot}} \geq x_{\min} \wedge x_{\text{robot}} \leq x_{\max}$ (similarly for the y-coordinate).
- Avoidance property : $x_{\text{robot}} \neq x_{\text{adversary}} \vee y_{\text{robot}} \neq y_{\text{adversary}}$.

The first scenario in our presentation involves the adversary patrolling on a specific route, while in the second the adversary is always moving towards the robot. Trajectory videos for both scenarios are available online[9].

7.1.1 Real Coordinates. Fig. 6 shows three possible trajectories that were generated after running the synthesized solution for 1000 turns against the patrolling adversary. Both robots move in the arena using rational coordinates in a 5x5 box. The initial location for the robot is the point $(0.5, 0.5)$ and the adversary begins its route from $(0.8, 0.8)$. While the adversary has a predetermined route, the robot is allowed to move towards any possible direction (vertically, horizontally and diagonally). Moreover, the robot can move at varying distances up to 0.1 units away from its current position, in both axes (i.e., $|x_{\text{robot}} - x'_{\text{robot}}| \leq 0.1$, and similarly for the y axis). Fig. 6a indicates how the synthesized solution can respond in a random pattern, covering different parts of the bounded arena while preserving safety. Fig. 6b and 6c demonstrate the resulting trajectories when the user introduces bias in the values returned by the random number generators, using the same generated witness from AE-VAL. As a result, the robot was limited to moves that would retain its position within the central area of the arena (Fig. 6b) and close to the bottom left corner of the patrolling adversary's route (Fig. 6c).

7.1.2 Integer Coordinates. For this experiment, we aimed to demonstrate the advantages that randomness can provide with respect to how well a robot covers a bounded arena, inspired by work in coverage path planning problems. Fig. 7 shows how two trajectories evolved over several turns (100, 250 and 1000 turns) for a similar motion planning problem using integer coordinates. To demonstrate which parts of the arena the robot explored we outline its trajectory with a bold black line, while the red line represents the trajectory of the adversary. In this game, the adversary is aggressively chasing after the robot in a random fashion. The robot's objective remains the same, i.e., move within the bounded arena while avoiding the adversary. The robot's initial location is the point $(0, 0)$, while the adversary begins at $(6, 6)$.

In fact, Fig. 7a, 7c, and 7e show moves performed by a random controller, while Fig. 7b, 7d,and 7f depict the behavior of the deterministic solution provided by the standard synthesis algorithm in JSyn-vg. It is apparent that the former visits 100% states in less than 250 turns, whereas the latter visits only 30% states in 1000 turns. This comparison showcases the advantages that a random solution can provide in terms of overall coverage as well as the diversity of behaviors that can be observed and exercised when an implementation can be generated that always considers the entire set of safe choices, instead of an instantiated strategy.

8 EVALUATION – SYNTHESIS TIME

Our case study in robot motion planning was inspired by results in this context from the most recent and related work on DT-Synth [35]. This reactive synthesis framework incorporates learning techniques to generate winning sets for infinite-state safety games in the form of decision trees. DT-Synth has been shown to outperform previous proposed synthesis tools, both in infinite-state (ConSynth [6] and finite-state problems (RPNI-Synth and SAT-Synth [36]). While the authors do not explicitly talk about randomness, the winning sets provided by DT-Synth are sufficient to generate implementations with diverse behavior. Despite this fact, the generated winning sets are subsets of the greatest fixpoint of safe states, which would lead to implementations that only exercise a fragment of the reachable state space. An example is the winning set that we mentioned for the motivating example in Section 3.

Note that DT-Synth works only for finite-branching game graphs, and the user must additionally specify a minimum value for the number of successors for each vertex in the graph. An incorrect value for this threshold can lead to unsound witnesses. With our JSyn-vg, such additional knowledge is not required from the user since it is only reliant on the original specification and is guaranteed to provide sound results, thanks to Theorem 4.5.

Table 2 presents the comparison of JSyn-vg and DT-Synth. As an addendum we included the synthesis times for the problems using the existing deterministic synthesis algorithm in JSyn-vg. As we mentioned in Sect. 5, the performance is identical when compared to synthesizing random witnesses.

For the purposes of this comparison, we used the infinite-state benchmarks presented in the original paper on DT-Synth [35], as well as the simulated avoidance games from Sect. 7.1, namely *bounded_evasion* and *bounded_evasion_ints*. The two tools have similar performance for half the benchmarks, with significant differences for the rest. For *diagonal*, the main distinction is that DT-Synth requires the definition of two additional expressions to guide the learning procedure, whereas JSyn-vg finds a solution without additional templates. On the other hand, DT-Synth's ability to synthesize memoryless strategies allows for faster synthesis for *solitarybox*, where the robot is simply moving freely within an infinite arena while staying within a horizontal stripe of width equal to three. JSyn-vg is targeted to synthesis of stateful systems, and as such, a more elaborate strategy is generated.

In the case of *repair-critical*, DT-Synth appears to be more efficient in terms of handling disjunctive expressions in the specification, while for *synth-synchronization* DT-Synth seems to require more elaborate hypotheses in order to come up with a witness.

[9] Pictures and videos of the simulated games presented in this section were anonymized and made available at https://figshare.com/s/ce2dfd885b3caf20f46d.

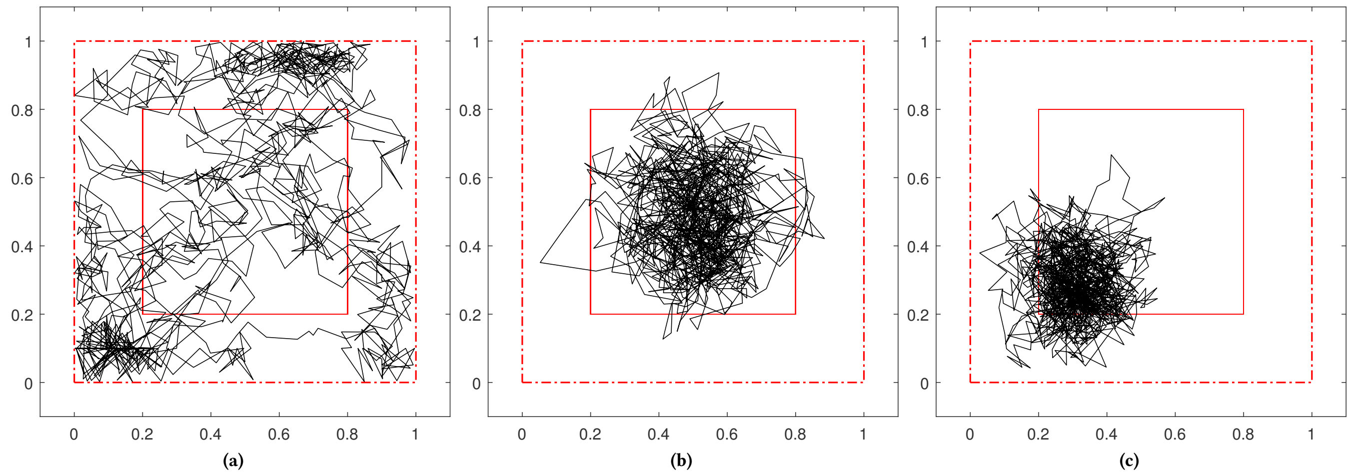


Figure 6: Random trajectories of a robot (irregular solid line) while avoiding a patrolling adversary (inner square).

Table 2: Synthesis time of DT-Synth and JSyn-vg (seconds).

Benchmark	JSyn-vg	JSyn-vg (random)	DT-Synth
box	0.603	0.606	**0.258**
diagonal	1.109	**1.011**	6.027
evasion	0.705	0.605	0.660
follow	3.34	3.029	**1.034**
limitedbox	3.229	3.332	3.350
solitarybox	1.902	1.816	**0.284**
square	5.823	**5.605**	6.44
program-repair	3.122	3.638	**2.452**
repair-critical	83.891	88.073	**30.593**
synth-synchronization	23.013	**23.2**	89.804
cinderella ($c = 2$)	20.061	**20.167**	> 900
cinderella ($c = 3$)	12.02	**11.294**	> 900
bounded_evasion	49.528	**49.662**	unsupported
bounded_evasion_ints	31.614	**32.611**	> 900

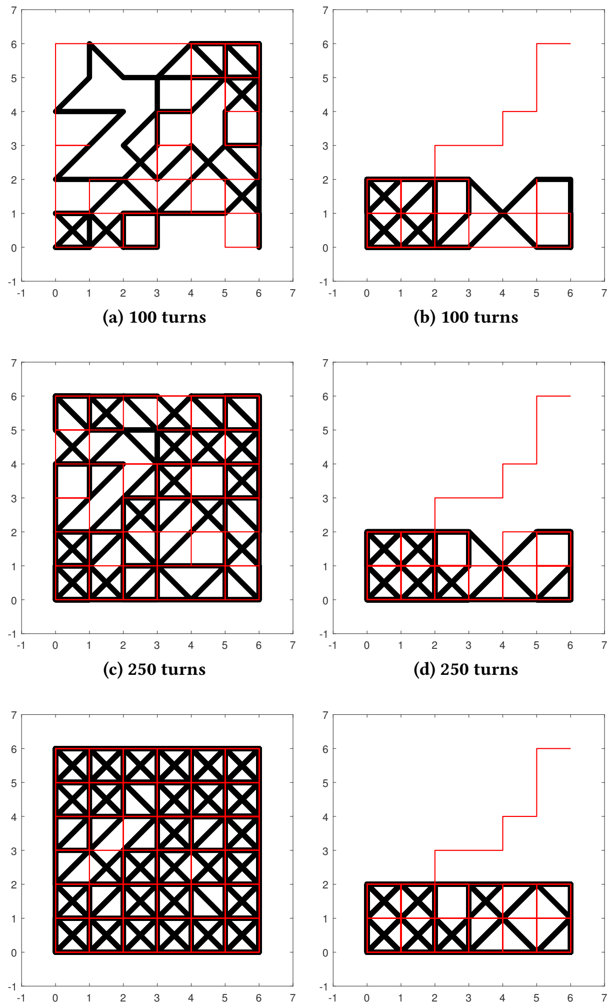
The latter is further demonstrated in the results for the *cinderella* and *bounded_evasion_ints* games, where DT-Synth fails to synthesize a witness within the timeout of 15 minutes. In contrast, JSyn-vg computes a greatest fixpoint of safe states and synthesizes a solution in a few seconds. Finally, for the game *bounded_evasion*, DT-Synth does not currently support the theory of linear real arithmetic.

9 RELATED WORK

The idea of synthesizing reactive designs with random behavior is relevant to synthesis for permissive games. This area has been explored in the past for finite-state problems [10, 32]. More recently, Fremont and Seshia described a formal extension to the theory of Control Improvisation to support reactive synthesis [20]. Their probabilistic approach is limited to finite-state problems and practically useful only for the subset of safety games. The end result is a maximally-randomized finite word generator, called an improviser, where each word satisfies the predetermined probability threshold constraints. In comparison, our approach synthesizes designs for infinite-state problems that simulate randomness. Furthermore, we do not provide guarantees regarding the randomness of the responses from the synthesized witness. Instead, we focus on synthesizing witnesses that consider regions of values as candidates to variable assignments, a problem reducible to SMT. In our case, the end product is an implementation that can be further refined by the engineer with a custom probability distribution to retrieve random values. This provides invaluable freedom as the user can then choose whether to express bias through the requirements or through the random number generators themselves.

The original work on JSyn-vg targeted the area of infinite-state problems. In this context, Beyenne et al. first proposed a template-based approach called ConSynth, where the specification is accompanied by a template regarding the shape of the solution to guide the synthesizer towards a solution [7]. In contrast, JSyn-vg is a completely automated approach exempting the user from necessity to further reason about the shape of the computed solution and allowing to focus on expressing the problem in the form of input-output contracts. Permissive solutions for infinite-state games have primarily been proposed in the context supervisor synthesis [13, 43], where a controller is synthesized considering a formal representation of the behavior (inputs) provided by the participating plant. Compared to this work, our proposed solution explores the applicability of synthesized controllers with random behavior, while the overall synthesis task is inherently harder due to not requiring an exact implementation of the controller's environment.

Neider and Markgraf recently proposed DT-Synth [35], a learning-based approach to synthesizing winning sets for infinite state games in the form of decision trees, as an extension to previous work by Neider and Topcu for finite-state problems [36]. DT-Synth requires additional knowledge regarding the number of successor states, where lack thereof can lead to unsound results. Similarly to ConSynth, for more complex problems, the game specification is supported by additional syntactic expressions that help the learner



(a) 100 turns (b) 100 turns

(c) 250 turns (d) 250 turns

(e) 1000 turns (f) 1000 turns

Figure 7: Trajectories and area coverage over time of random (left column) versus deterministic (right column) controller.

converge faster to a solution. In contrast, our Skolem extraction algorithm is guaranteed to provide sound witnesses and does not depend on additional user-provided input.

10 CONCLUSION

We have presented a novel Skolemization procedure for the AE-VAL solver enabling the synthesis of infinite-state reactive implementations with random behavior. The proposed solution is an extension to the synthesis algorithm JSyn-vg that computes a greatest fixpoint of safe states. The product is a witness where values inside safe regions are being considered as equally safe candidate assignments. Our solution provides the engineer with flexibility when it comes to introducing additional bias through the specification or the implemented random number generators.

To the best of our knowledge, this is the first work that is capable of synthesizing random infinite-state systems. We showed how the extended synthesis framework can be effectively used to synthesize promising solutions in the context of robot motion planning, as well as a novel application in fuzz testing. In the future, we wish to continue exploring the area of reactive fuzzer synthesis, particularly in the context of identifying a formal specification standard. To expand its applicability in robot motion planning, we wish to explore ways to support liveness specifications, as well as soft requirements. The outstanding result of this work is a Skolem extraction procedure general enough to be applicable to other, unexplored, aspects of the synthesis problem.

ACKNOWLEDGMENTS

The authors would like to thank Kristopher Cory for his invaluable advice towards setting up the fuzz testing experiments. The authors would also like to thank the anonymous reviewers for their comments. This work was funded in part by the United States Department of the Navy, Office of Naval Research under contract N68335-17-C-0238.

REFERENCES

[1] [n.d.]. Peach fuzzer. https://www.peach.tech/.
[2] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2017. Automated formal synthesis of digital controllers for state-space physical plants. In *International Conference on Computer Aided Verification*. Springer, 462–482.
[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.
[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.
[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.
[6] Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *POPL*. ACM, 221–233.
[7] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *POPL*. ACM, 221–234.
[8] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *LPAR (short papers) (EPiC Series in Computing)*, Vol. 35. EasyChair, 15–27.
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[10] Patricia Bouyer, Marie Duflot, Nicolas Markey, and Gabriel Renault. 2009. Measuring permissivity in finite games. In *International Conference on Concurrency Theory*. Springer, 196–210.
[11] Supratik Chakraborty, Dror Fried, Lucas M Tabajara, and Moshe Y Vardi. 2018. Functional synthesis via input-output separation. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–9.
[12] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 model checker. In *International Conference on Computer Aided Verification*. Springer, 510–517.
[13] Koen Claessen, Jonatan Kilhamn, Laura Kovács, and Bengt Lennartson. 2017. A Supervisory Control Algorithm Based on Property-Directed Reachability. In *Haifa Verification Conference*. Springer, 115–130.
[14] Grigory Fedyukovich and Aarti Gupta. 2019. Functional Synthesis with Examples. In *CP (LNCS)*, Vol. 11802. Springer, 547–564.
[15] Grigory Fedyukovich, Arie Gurfinkel, and Aarti Gupta. 2019. Lazy but Effective Functional Synthesis. In *VMCAI (LNCS)*, Vol. 11388. Springer, 92–113.
[16] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2015. Automated Discovery of Simulation Between Programs. In *LPAR (LNCS)*, Vol. 9450. Springer, 606–621.
[17] Bernd Finkbeiner. 2016. Synthesis of Reactive Systems. *Dependable Software Systems Engineering* 45 (2016), 72–98.
[18] Dustin Fraze. [n.d.]. The DARPA Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge.

[19] Daniel J Fremont, Alexandre Donzé, and Sanjit A Seshia. 2017. Control improvisation. *arXiv preprint arXiv:1704.06319* (2017).

[20] Daniel J. Fremont and Sanjit A. Seshia. 2018. Reactive Control Improvisation. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 307–326.

[21] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. 2018. The JKind Model Checker. In *International Conference on Computer Aided Verification*. Springer, 20–27.

[22] Andrew Gacek, Andreas Katis, Michael W Whalen, John Backes, and Darren Cofer. 2015. Towards Realizability Checking of Contracts Using Theories. In *NFM (LNCS)*, Vol. 9058. Springer, 173–187.

[23] Enric Galceran and Marc Carreras. 2013. A survey on coverage path planning for robotics. *Robotics and Autonomous systems* 61, 12 (2013), 1258–1276.

[24] Sumit Gulwani. 2010. Dimensions in program synthesis. In *PPDP*. ACM, 13–24.

[25] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[26] Falk Howar and Bernhard Steffen. 2018. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*. Springer, 123–148.

[27] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. [n.d.]. The Lustre V6 Reference Manual. http://www-verimag.imag.fr/Lustre-V6.html.

[28] Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W Whalen. 2018. Validity-guided synthesis of reactive systems from assume-guarantee contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 176–193.

[29] Andreas Katis, Andrew Gacek, and Michael W Whalen. 2015. Machine-Checked Proofs For Realizability Checking Algorithms. In *VSTTE*. Springer, 110–123.

[30] Andreas Katis, Andrew Gacek, and Michael W Whalen. 2016. Towards synthesis from assume-guarantee contracts involving infinite theories: a preliminary report. In *FormaliSE*. IEEE, 36–41.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[32] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. 2017. Shield synthesis. *Formal Methods in System Design* 51, 2 (2017), 332–361.

[33] Newton Lee. 2015. DARPA's Cyber Grand Challenge (2014–2016). In *Counterterrorism and Cybersecurity*. Springer, 429–456.

[34] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[35] Daniel Neider and Oliver Markgraf. 2019. Learning-based synthesis of safety controllers. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 120–128.

[36] Daniel Neider and Ufuk Topcu. 2016. An automaton learning approach to solving safety games over infinite graphs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 204–221.

[37] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. 2017. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 301–318.

[38] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 543–553.

[40] Michael Rash. [n.d.]. afl-cov - AFL Fuzzing Code Coverage. https://github.com/mrash/afl-cov.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[42] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic device driver synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 73–86.

[43] Mohammad Reza Shoaei, Laura Kovács, and Bengt Lennartson. 2014. Supervisory control of discrete-event systems via IC3. In *Haifa Verification Conference*. Springer, 252–266.

[44] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[45] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.

[46] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.

[47] Michal Zalewski. [n.d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.