Synthia: a Generic and Flexible Data Structure Generator

Marc-Antoine Plourde
Université du Québec à Chicoutimi
Canada

Sylvain Hallé
Université du Québec à Chicoutimi
Canada

ABSTRACT

Synthia is a versatile, modular and extensible Java-based data structure generation library. It is centered on the notion of "pickers", which are objects producing values of a given type on demand. Pickers are stateful and can be given as input to other pickers; this chaining principle can generate objects whose structure follows a complex pattern. The paper describes the core principles and key features of the library, including test input shrinking, provenance tracking, and object mutation.

KEYWORDS

synthetic data generation, fuzzing, test reduction

ACM Reference Format:
Marc-Antoine Plourde and Sylvain Hallé. 2022. Synthia: a Generic and Flexible Data Structure Generator. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3510454.3516834

1 INTRODUCTION

The problem of generating synthetic data that mimics a real-world counterpart can find a purpose in many situations. A long-standing line of research has focused on *fuzzing*, which is the generation of (random or structured) data fed to a system component with the goal of discovering inputs causing crashes or failures [7]. Artificial data also proves useful outside bug finding: for instance, it can be used to fake the operation of a system component, forming the basis of a development aid called a *stub* or *mock object* [9]. In addition, contrary to real-world data, synthetic inputs can be generated according to a number of configurable parameters, making it possible to perform *controlled experiments* by varying these parameters and measuring their effect on some aspect of a system [8].

This paper presents Synthia, a Java library for the generation of synthetic inputs of various kinds. It was built with three design goals. The first is **versatility**: the library collates under a uniform framework functionalities that are typically handled by distinct tools. Objects implementing Synthia's `Picker` interface can generate scalars, composite data structures, or sequences of objects according to a formal grammar or a Markov chain, among others. Besides classical fuzzing, Synthia supports automated test input shrinking, object mutation, provides a framework for monkey testing reactive systems, and implements a rudimentary form of provenance tracking. Variants of these pickers make it possible to use them as stateful mock objects.

The second is **modularity**: most `Picker` objects in the library are instantiated by passing instances of other pickers which they use as sources of "choices". Thus, an arbitrarily complex chain of pickers can serve as the input of another one, and result in drastically different objects of a given type being produced depending on the precise way these pickers are passed to each other. This distinctive feature gives its name to the library: it follows the operation of analog *synth*esizers of the 1970s, which used signal generators and modules connected by patch cables to produce various sounds.

Finally, the library also favors **extensibility**: its core is formed of a small number of elementary pickers, which are complemented by domain-specific extensions. Moreover, a user can create its own picker objects as Java classes following the `Picker` interface, and these pickers can then be used along the other pickers provided by the library. Synthia is implemented in Java and is available under an open source license.[1] Its current version consists of roughly 5,300 LOC, comes with a complete API documentation and includes dozens of code examples.

2 FUNDAMENTAL CONCEPTS

In the following, we describe the core principles of Synthia.

The Picker Interface. The fundamental object in Synthia is called a *picker*. Its basic task is to return (i.e. "pick") an object of given type T every time it is asked. To this end, the Java interface `Picker<T>` declares a no-args method called `pick`; it is up to each concrete picker class to implement whatever logic is required to produce the said object.

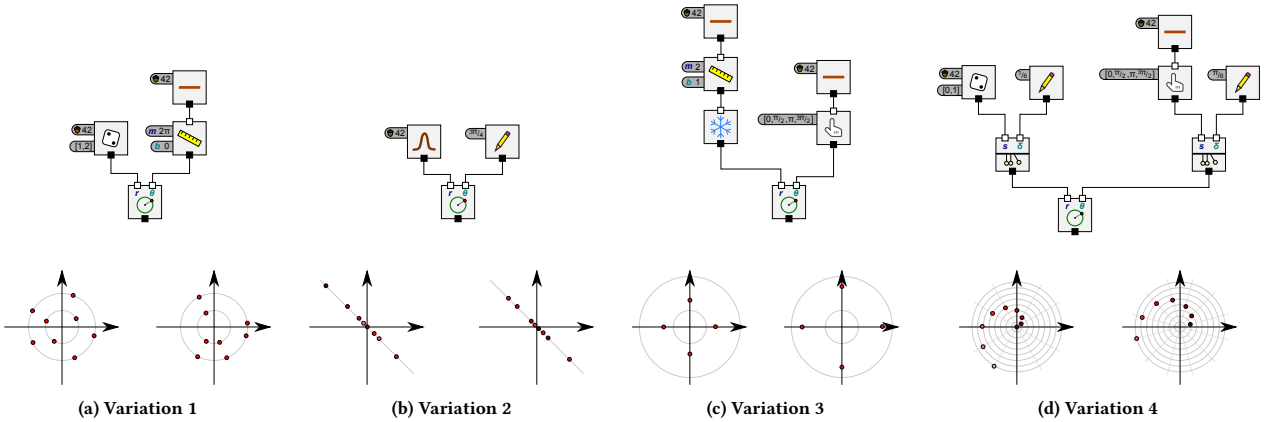
It is important to note that the use of a picker does not entail the presence of randomness. As we shall see, a picker for integers may return a pseudo-randomly generated number, but may also produce a sequence of outputs following a regular pattern, or even a single constant value. In addition, pickers are *stateful*: the output they produce may depend on other outputs returned on previous calls to `pick`. Hence, the `Tick` picker returns a number that increments by a given amount upon each successive call, and acts as a form of counter; the `Playback` picker iterates over a list of predefined values, and loops around once the list is exhausted.

Because of their stateful nature, every picker is required to implement two additional methods called `reset` and `duplicate`. Method `reset` restores the picker to its uniquely-defined initial state. This means that successive calls to `pick` from this point on reproduce the same sequence of values as when the object was constructed.[2] Method `duplicate` creates a copy of the picker; this copy can either be stateless or stateful. A stateless copy amounts to creating

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516834

[1] https://github.com/liflab/synthia
[2] This trait can be declared for other objects with an interface called `Resettable`, which will be mentioned in Section 4.



(a) Variation 1 (b) Variation 2 (c) Variation 3 (d) Variation 4

Figure 1: Multiple ways of parameterizing a `HyperspherePicker`, and the possible sets of values each can produce.

a new instance obtained by calling the constructor with the same arguments as the original; on its side, a stateful copy places the new instance in the same internal state as the current state of the original. Note that this requirement applies to all pickers, including those generating pseudo-random numbers. Pseudo-random pickers implement an additional interface called `Seedable`, declaring a method called `setSeed` that can be used to explicitly define the seed value used as the starting point of the generation.

The core Synthia library provides more than a dozen pickers for producing commonly used values, such as Booleans, numbers and strings. `PickElement` picks an element from a set based on probabilities associated to each element. Synthia also provides pickers for generating common composite data structures. `PickList` generates a list of elements of type T; `PickSet` and `PickArray` do the same for sets and arrays, respectively. `PickTree` produces a tree by using other pickers for selecting its depth, node degree and node contents. `HypercubePicker` generates a point within a hypercube of given dimensions, and `HypershperePicker` generates an n-dimensional point on a hypersphere, given a radius r and $n-1$ angles.

The library also includes a few pickers producing elements that simulate some behavior. `MarkovChain`, as its name implies, performs a walk in a Markov chain and, upon each transition, produces an object by asking the picker associated to the state it reaches. `BehaviorTree` does the same for a path in a behavior tree. `GrammarPicker` generates sequences of symbols picked from a set defined by a context-free grammar.

Finally, some pickers have special behavior that can be useful: `Constant` always returns the same predefined value; `Nothing` immediately throws a `NoMoreElementException` when asked for a value; `Freeze` requests a value from another picker a single time, and then repeatedly returns this value; `Playback` returns a predefined sequence of values by iterating over a fixed list. `Scramble` asks for a variable number of values from another picker, shuffles their ordering and plays them back in that modified order. Finally, `PickIf` acts as a form of filter: upon a call to `pick`, it repeatedly asks for values from another picker, until one is found that satisfies an arbitrary user-defined condition.

In addition to the pickers already provided by Synthia, users can define custom ones by simply writing a new class implementing the `Picker` interface. Once defined, this picker can be passed as the argument of other pickers.

Wiring Pickers. A core feature of Synthia is the fact a picker may be instantiated by passing to it one or more other pickers. Intuitively, this represents the fact that this picker must make a number of "choices" when producing an output object, and that these choices are based on the values produced by other pickers when they are queried. Thus, a given generator for a set of values may be defined as a form of "wiring diagram", making explicit the way in which pickers are composed. Given that each picker is deterministic (including pseudo-random generators for a given seed), the wiring diagram therefore unambiguously defines the exact sequence of values produced by successive calls to the downstream picker, favoring the reproducibility of experiments that are conducted on a synthetic dataset.

In order to illustrate the flexibility offered by Synthia in generating even simple objects, consider the diagrams shown in Figure 1, which show scenarios where two-dimensional points are produced by an `HyperSpherePicker`, represented at the bottom of each figure.[3] What differs is how the two sources required by this picker (one for a radius r and another for an angle θ) are created. For each point, Variation 1 selects a radius of either 1 or 2. A random floating-point x between 0 and 1 is also selected, to which the affine transform $2\pi x + 0$ is then applied. The end result is that each point lies at any angle between 0 and 2π on the circle of radius 1 or 2. The two plots at the bottom represent the first few possible values one could obtain by starting from a different random seed.

The remaining diagrams show the effect of wiring the `HyperspherePicker` in a different way. Variation 2 fixes the angle to $3\pi/2$, and picks the radius according to a normal distribution, resulting in points clustered close to the origin. Variation 3 shows the use of the `Freeze` picker: a random float value between 1 and 3 is picked for the radius, and this radius will then be used for all points produced

[3] Each picker class is represented by a distinct pictogram; the legend for each pictogram is available in the online API documentation (https://liflab.github.io/synthia/javadoc).

```
RandomInteger ri = new RandomInteger(0, 2).setSeed(42);
Constant<Float> c1 = new Constant<>(1 / 6f);
Tick radius = new Tick(ri, c1);
RandomFloat rf = new RandomFloat().setSeed(42);
Choice<Double> start = new Choice<>(rf).add(0d, 0.25)
  .add(PI / 2, 0.25).add(PI, 0.25).add(3 * PI / 2, 0.25);
Constant<Float> c2 = new Constant<>(1 / 6f);
Tick angle = new Tick(start, c2);
HyperspherePicker hp = new HyperspherePicker(radius, angle);
for (int i = 0; i < 100; i++) {
  Utilities.print(System.out, hp.pick());
}
```

Figure 2: The source code producing the points in Figure 1d.

by the `HyperspherePicker`. The angle is randomly selected among four values lying at intervals of $\pi/2$. As a result, different seeds will produce sets of points lying at a different distance from the origin, but the same distance for each picker instance. Finally, Variation 4 shows the use of the `Tick` picker. The radius of the first point is either 0 or 1, and each successive radius increments by a fixed amount of $1/6$. The starting angle is randomly selected between four values, and each successive angle is incremented by a fixed amount of $\pi/6$; this results in a spiral pattern with slight variations depending on the starting seed. Each of the diagrams above corresponds to less than 10 lines of Java code.[4] Figure 2 shows the code for Variation 4.

This simple example involved only a few basic pickers; however one can see the potentially complex objects and patterns one can create by composing existing and custom pickers.

3 ADVANCED FEATURES

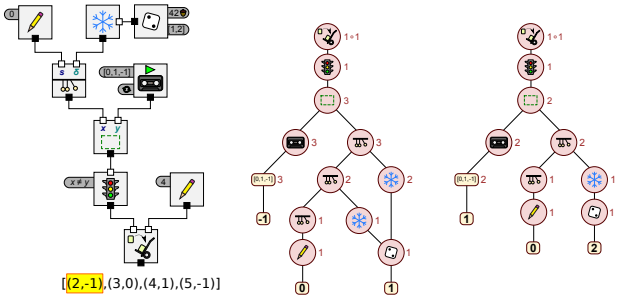
We now enumerate some of the distinctive functions of the library.

Explainability. Synthia incorporates functionalities for explaining the values produced by a chain of pickers. It leverages an existing library called *Petit Poucet* [4], which allows one to point to a part of an output produced by some object, and to retrace it back to parts of the inputs that contributed to the production of that value. In the terminology of Petit Poucet, a *designator* is an object used to refer to a particular part of the input or output of a given computation. A relationship between a part of an output and one or more parts of the input is called an *explanation*. An object that can provide such an explanation implements an interface called `Queryable`, which declares a single method called `query`.

In Synthia, each value produced by a picker is uniquely designated by its index in the sequence of successive calls to `pick`. If a picker produces a given output object based on values provided by other (upstream) pickers, it can link this output to the indices of the corresponding inputs. This process can be repeated all the way up to the roots of the chain of connected pickers, resulting in an *explanation graph* retracing the precise values of each picker along the path that contributed to the generation of a given object.

Figure 3 shows an example of this feature. On the left, a chain of pickers is instructed to generate a list of four two-dimensional points using a `PrismPicker`, with constraints on their possible values. The x coordinate starts at 0 and increments by a fixed value

[4] API documentation and code examples: https://liflab.github.io/synthia/javadoc/



(a) Wiring (b) Explanation (c) Other explanation

Figure 3: A wiring of pickers producing a list of four points, and two explanation graphs for the first point in the list.

that is initially chosen to be either 1 or 2; the y coordinate repeatedly iterates through the values 0, 1 and -1; a `PickIf` removes any point lying on the line $x = y$. A possible first output of this chain is the list shown at the bottom. Explanations are fine-grained: a graph can be built for a given *part* of this output, such as the first pair of numbers inside the list. One can then ask for an explanation of the first point inside the list by passing the designator $1 \circ 1$ (meaning "the first element of the first output") to the picker through a call to a special method named `query`. This picker links this value to the output produced by other pickers, ultimately producing the graph shown on Figure 3b.

Each node in the graph corresponds to one picker from the chain on the left, and next to each is written the designator corresponding to one of their output values. By examining this graph, one can see that the first point of the list is actually the third generated by the `PrismPicker`; the first two (the points $(0, 0)$ and $(1, 1)$) having been rejected by the filter. Since the `Tick` picker produces a value by incrementing the previous one, the third output of `Tick` depends on the second one and the second value produced by `Freeze`. As expected, all values produced by this picker emanate from a single pick from `RandomInteger`. Leaves of this graph are labeled with the concrete values produced by a given picker. Note how this structure is dynamic and depends on the actual values produced; Figure 3c shows an alternate explanation, which is what one obtains when `RandomInteger` picks the value 2 instead of 1.

This graph is not a static template obtained from the wiring of pickers; to illustrate this fact, Figure 3c shows an alternate explanation, which is what one obtains when `RandomInteger` picks the value 2 instead of 1; the output list becomes $[(2, 1), (4, -1), (6, 0), (8, 1)]$. Note how the explanation graph has a slightly different structure, and that the relative indices of output values in some pickers are not the same.

Bounded Pickers and Enumerations. By default, most pickers are assumed to be *unbounded*: they can always return a value on each call to `pick`. Synthia also provides an interface called `Bounded<T>`, which declares a method called `isDone`. A bounded picker acts like a classical `Iterator`: it produces output values until a call to `isDone` returns false. Some pickers are bounded by definition, such as `Playback` and `PickUntil`. An alternate way of obtaining a bounded picker is by wrapping an unbounded picker inside a

Bound object; this picker first selects a number of elements, and then relays values obtained from another picker until the bound is reached.

Since bounded pickers can be assimilated to a finite enumeration, combinations of values for multiple such pickers can also be enumerated. This is the task of the `Enumerate` picker: given a list of n pickers (which may be of different output types), calls to `pick` will produce an array of n values (one from each input picker) such that all combinations of values from each picker eventually occurs. This is possible due to the property that calling `reset` on a bounded picker once `isDone` returns true restarts its sequence of values from the beginning, and in the same order.

The use of `Enumerate` makes it possible to easily generate values for a test procedure that exhaustively explores the value space induced by multiple input parameters. If connected to the input of a `Scramble` picker, one can also make sure that the combinations are enumerated in a (pseudo-)random fashion. As unbounded pickers can be turned into bounded ones, the opposite operation is also possible. A bounded picker p can be wrapped into a picker called `Unbound`. This picker simply relays the output values of p, until a call to `isDone` returns false. From this point on, subsequent calls to `pick` select one of the values returned by p in the past.

Knitting. A special picker called `Knit` makes it possible to interleave the output of multiple pickers. The `Knit` picker must be instantiated by passing to it a picker of pickers p (that is, a picker producing `Picker<T>` objects). Upon every call to `pick()`, `Knit` proceeds as follows. First, it flips a (biased) coin to decide whether to create a new instance of `Picker<T>`; if so, it calls p`.pick()` and adds the resulting picker instance to its set of "living" pickers. Then, it selects one of the living pickers, and returns the object resulting from a call to `pick()` on that picker. If this picker cannot produce a new value (e.g. it throws a `NoMoreElementException`), it is considered "dead" and is removed from the set of living pickers. In such a case, `Knit` flips a coin to decide whether to create a new instance of `Picker<T>`; if so, it calls p`.pick()` and adds the resulting picker instance to its set of "living" pickers, and the process repeats. This picker is especially useful to generate sequences of events produced by multiple entities that follow an independent lifecycle. For example, a `MarkovChain` picker could be configured to simulate possible sequences of pages visited by a user in a website; passing it to `Knit` results in an interleaved sequence of page requests for multiple visitors, similar to a real-world web server log. A variant of `Knit` has been used to simulate the trajectories of two-dimensional balls in a virus contagion simulator.[5]

4 TESTING FACILITIES

Given its nature as a data structure generator, a natural use case for Synthia is as a provider of test inputs. To this end, the library provides a few facilities geared towards its use in a testing context.

Object Mutation. Some classes defined in Synthia are called *mutators*: they are pickers that receive an object, apply an elementary transformation on it, and return the "mutated" object. Mutators can be appended to a chain of pickers producing valid inputs, and turn them into invalid ones by slightly altering them. What mutation is performed, and how it is applied may, again, depend on values produced by other pickers.

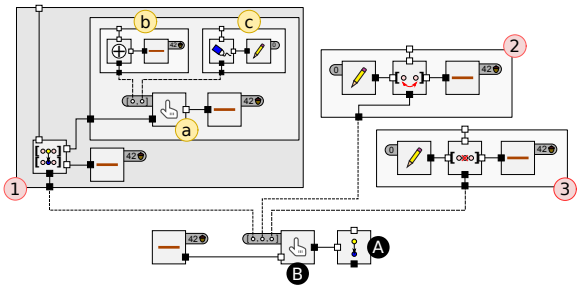


Figure 4: Instantiating a mutator for a list.

A simple example is the `Offset` picker, which receives a number, and shifts its value by an amount specified by the output of another picker, thereby simulating noise or uncertainty. For instance, it can be used to insert some amount of noise or uncertainty in a source of otherwise precise values. Another simple mutator is `Replace`: when it receives an object, it outputs instead a value produced by another picker; its usefulness will be made clearer in later examples. Synthia also provides mutators for composite data structures, such as lists. `DeleteElement` receives a list and selects an element to remove; `InsertElement` chooses a position where a value provided by another picker is to be inserted; `Swap` selects two elements and inverts their position in the list. In addition, some mutators take as one of their parameters a picker producing another mutator. This is the case of the `Mutate` picker: it asks for an object o and a mutator m, and returns the application of m on o. The same goes for `MutateElement`: for a list where elements are of given type T, the picker is given an arbitrary mutator m for T, selects an element e of the input list, and replaces it by the application of m on e.

Figure 4 shows an example of the flexibility of this design. The diagram culminates into a `Mutate` picker (box A) that receives as one of its inputs an arbitrary list of numbers (not shown). The picker that provides a mutator for each list is a `Choice` (box B), which may pick one of three possible mutators, represented by boxes 1–3. The first is an instance of `MutateElement`, whose mutator picker is another `Choice` (box a), selecting either an `Offset` (box b) or a `Replace` feeding the constant value 0 (box c). The other two list mutators are instances of `Swap` (box 2) and `DeleteElement` (box 3). All these pickers use a distinct `RandomFloat` as their source of randomness for making their respective choices. The end result is that, for every list given to the picker of box A, three equally probable outcomes are possible: (1) one element is chosen (box 1), and, with equal probability (box a), is either offset by a random value in $[0, 1]$ (box b), or replaced with 0 (box c); (2) the first element is swapped with another one (box 2); (3) one element is deleted (box 3).

Test Reduction. In addition to generating test inputs (either from random sources or otherwise), the library can also be used to search for simpler inputs once a failing test case has been found, following the *shrinking* principle found in the QuickCheck framework [1] and its Java derivatives like JUnit-QuickCheck [5]. An interface called `Shrinkable` extends the basic `Picker` interface by providing

[5] https://github.com/sylvainhalle/virus-contagion

an additional method called `shrink`. Each shrinkable picker p implementing `Shrinkable` assumes a certain partial order $\sqsubseteq$ over the elements of T. If the picker originally returns elements from a domain $T' \subseteq T$, for a given $t \in T$, a call to `p.shrink(t)` returns a new picker instance p'; it is not necessarily an instance of the same class as p, but it guarantees that its values are in $\{t' \in T' : t' \sqsubset t\}$. By virtue of composition, objects that are generated through a wiring of `Shrinkable` pickers can be shrunk automatically.

This mode of shrinking is different from QuickCheck implementations in a fundamental aspect: it is the *pickers* that are shrinkable, and not the values they produce; thus, `shrink` returns a new *picker*, and not a (finite) collection of pre-shrunk values. At any moment, this picker can be used as a drop-in replacement for the original picker without any other modification to the setup. Moreover, the picker truncates the domain of the original picker, but otherwise preserves its properties. For example, if a picker p produces odd numbers, a call to `p.shrink(15)` will result in a picker instance that still produces odd numbers, and not just *any* number smaller than 15. In the same way, if a `GrammarPicker` p produces a sequence of objects s according to a BNF grammar, then `p.shrink(`s`)` produces a picker that returns sub-sequences of s that are still valid according to the grammar, and not just any sub-sequence of s. This characteristic extends to any shrinkable picker.

In addition, the shrinking process propagates: a picker that produces a value depending on the output of another picker may request a shrunk version of this picker upon its own call to `shrink`. Consider a picker producing a list whose length is provided by applying $2x + 1$ to the integer value produced by another picker. Upon a call to `shrink` with the list $[3, 1, 4, 1, 5, 9, 2]$, of length 7, this picker will request from its upstream source a shrunk instance producing values lower than $(7 - 1) \div 2 = 3$. Note that this will result in lists whose length y still follows the relation $y = 2x + 1$, but having fewer than 7 elements.

This design still provides flexibility in how the shrunk version of the picker is produced. Upon a call to `pick(t)`, simple pickers, such as one for integers in the range $[a, b]$, may merely return a new instance of the same class with different parameters (e.g. the range $[a, t]$). In other cases, the designer of a custom picker object may elect to return a `Playback` picker enumerating a fixed set of values, thereby reproducing the behavior of JUnit-QuickCheck. If for some reason, no "smaller" value than t can be obtained, a picker can also return an instance of `Nothing`. A picker p that does not implement `Shrinkable<T>` can also still be shrunk under some conditions. If T is a type that implements the Java `Comparable` interface, p can be wrapped within a `PickSmaller` picker. This picker provides a "poor man's" shrinking mechanism: given a reference element t, this picker repeatedly asks for an element t' from p until one is found such that $t' \sqsubset t$ (calculated using class T's method `compareTo`).

Test Automation. Finally, Synthia offers two classes to automate the test-and-shrink process on an abstract system. An interface called `Testable` declares a method called `test` taking an array of objects (the inputs of a test case) and returning a Boolean value (representing the test's verdict). An object called `Assert` is instantiated with the pickers that supply parameters, and a `Testable` object. It takes care of generating inputs until a combination causes `test` to return false; once this is done, a shrunk version of the original pickers is obtained, and the process repeats. This operation stops if the resulting pickers throw either a `GiveUpException` or a `NoMoreElementsException`. The last value causing the condition to fail can then be queried with `getShrunk`.

Contrary to existing QuickCheck implementations, which shrink a single test input, `Assert` can apply shrinking on test cases taking multiple input values. This can be done by encapsulating all input parameters within a `PickArray`, and using its built-in shrink feature. In such a case, a shrunken test input is any combination of inputs for which each parameter is equal or smaller to the corresponding parameter in the original failing test case.

A similar functionality is also available for *reactive* components, which are tested by performing a sequence of actions, each resulting in a change of state of the system. The `Action` interface defines a single method called `execute`, representing the abstract notion of submitting an "action" to a reactive system; for example, a possible concrete action could be a button click on a GUI widget. The `Monkey` object provided by Synthia takes as input a picker producing actions, and a `Resettable` object. It starts with a discovery phase, repeatedly picking actions from the and performing them one by one. As soon as an exception is caught, the sequence of actions performed so far is stored and the object under test is reset. The monkey then enters a shrinking phase, where shorter sub-traces of the original are repeatedly generated and tested. Whenever a shorter sequence still throws an exception, this sequence replaces the original, and the shrinking process restarts from this new reference.

5 CONCLUSION

In this paper, we have shown how Synthia allows the generation of synthetic data structures in a modular fashion. Its powerful functionalities rely on a handful of simple abstract interfaces (e.g. `Picker`, `Shrinkable`, `Testable`, `Resettable`, `Bounded` and `Action`), which can be implemented by classes in order to generate, mutate or test input data on an arbitrary object. Case in point, Synthia has been used in recent scientific publications to simulate conference program committees [3], produce sequences of events to illustrate an event stream processor [2], generate web pages with layout faults [6], and create synthetic logs to test a monitor [10].

REFERENCES

[1] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[2] S. Hallé. Explainable queries over event logs. In *EDOC*, pages 171–180. IEEE, 2020.

[3] S. Hallé. Computer simulations of scientific peer reviewing. *IEEE Access*, 9:111595–111607, 2021.

[4] S. Hallé and H. Tremblay. Foundations of fine-grained explainability. In *CAV*, volume 12760 of *LNCS*, pages 500–523. Springer, 2021.

[5] P. Holser. JUnit-Quickcheck. https://github.com/pholser/junit-quickcheck, Accessed October 13th, 2021.

[6] S. Jacquet, X. Chamberland-Thibeault, and S. Hallé. Automated repair of layout bugs in web pages with linear programming. In *ICWE*, volume 12706 of *LNCS*, pages 423–439. Springer, 2021.

[7] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.*, pages 1–1, 2019.

[8] T. J. Santner, B. J. Williams, and W. I. Notz. *The Design and Analysis of Computer Experiments*. Springer series in statistics. Springer, 2003.

[9] D. Spadini, M. F. Aniche, M. Bruntink, and A. Bacchelli. Mock objects for testing Java systems. *Empir. Softw. Eng.*, 24(3):1461–1498, 2019.

[10] R. Taleb, R. Khoury, and S. Hallé. Runtime verification under access restrictions. In *FormaliSE@ICSE 2021*, pages 31–41. IEEE, 2021.