Natural Symbolic Execution-Based Testing for Big Data Analytics

YAOXUAN WU, University of California at Los Angeles, USA
AHMAD HUMAYUN, Virginia Tech, USA
MUHAMMAD ALI GULZAR, Virginia Tech, USA
MIRYUNG KIM, University of California at Los Angeles, USA

Symbolic execution is an automated test input generation technique that models individual program paths as logical constraints. However, the realism of concrete test inputs generated by SMT solvers often comes into question. Existing symbolic execution tools only seek arbitrary solutions for given path constraints. These constraints do not incorporate the naturalness of inputs that observe statistical distributions, range constraints, or preferred string constants. This results in unnatural-looking inputs that fail to emulate real-world data.

In this paper, we extend symbolic execution with consideration for incorporating naturalness. Our key insight is that users typically understand the semantics of program inputs, such as the distribution of `height` or possible values of `zipcode`, which can be leveraged to advance the ability of symbolic execution to produce natural test inputs. We instantiate this idea in NaturalSym, a symbolic execution-based test generation tool for data-intensive scalable computing (DISC) applications. NaturalSym generates natural-looking data that mimics real-world distributions by utilizing user-provided input semantics to drastically enhance the naturalness of inputs, while preserving strong bug-finding potential.

On DISC applications and commercial big data test benchmarks, NaturalSym achieves a higher degree of realism —as evidenced by a perplexity score 35.1 points lower on median, and detects 1.29× injected faults compared to the state-of-the-art symbolic executor for DISC, BigTest. This is because BigTest draws inputs purely based on the satisfiability of path constraints constructed from branch predicates, while NaturalSym is able to draw natural concrete values based on user-specified semantics and prioritize using these values in input generation. Our empirical results demonstrate that NaturalSym finds injected faults 47.8× more than NaturalFuzz (a coverage-guided fuzzer) and 19.1× more than ChatGPT. Meanwhile, TestMiner (a mining-based approach) fails to detect any injected faults. NaturalSym is the first symbolic executor that combines the notion of input naturalness in symbolic path constraints during SMT-based input generation. We make our code available at https://github.com/UCLA-SEAL/NaturalSym.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Symbolic Execution, Naturalness, DISC Applications

ACM Reference Format:
Yaoxuan Wu, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2024. Natural Symbolic Execution-Based Testing for Big Data Analytics. *Proc. ACM Softw. Eng.* 1, FSE, Article 118 (July 2024), 24 pages. https://doi.org/10.1145/3660825

Authors' addresses: Yaoxuan Wu, University of California at Los Angeles, Los Angeles, USA, thaddywu@cs.ucla.edu; Ahmad Humayun, Virginia Tech, Blacksburg, USA, ahmad35@vt.edu; Muhammad Ali Gulzar, Virginia Tech, Blacksburg, USA, gulzar@cs.vt.edu; Miryung Kim, University of California at Los Angeles, Los Angeles, USA, miryung@cs.ucla.edu.

1 INTRODUCTION

Data-intensive scalable computing (DISC) applications have emerged as a necessity for large-scale data processing. DISC Frameworks such as MapReduce [26] and Apache Spark [69] provide interfaces for developers to manage and manipulate data across clusters at large scale. Despite the widespread adoption of these DISC frameworks, there still remain significant challenges associated with automated test input generation for DISC applications.

Symbolic execution is a popular white-box test generation technique [18, 28, 51]. Symbolic executors traverse program paths symbolically and collect the branch predicate along the executed path to model logical constraints corresponding to individual program paths. An SMT solver such as CVC5 [14] or Z3 [25] then produces concrete solutions that satisfy these path constraints. Test cases generated by symbolic execution are desirable due to their capability to enumerate individual program paths and increase code coverage in a systematic manner. Further, the test inputs generated by symbolic execution are often concise, as symbolic execution generates a single input that covers the equivalence class of each program path.

However, developers often complain that the inputs generated by SMT solvers appear *unrealistic* [30]. Suppose an application analyzes the average height of adults in California using a dataset that may include non-California residents and people born after 2005. One of the application's path constraints represents an equivalence class of inputs that map to non-California residents born before the year 2005 expressed as `birth_year<2005 && state_of_residence!="CA"`. For this path constraint, CVC5 generates a test input with `state_of_residence=""` and `birth_year=0`, despite the fact an empty string does not correspond to any state and a birth year of 0 is highly improbable. Such test data may be dismissed as irrelevant due to its implausibility; the highly unlikely nature of these inputs might lead developers to overlook potential bugs, as they appear less relevant to practical scenarios.

The root cause of unnaturalness is that existing symbolic executors construct path constraints only from branch predicates in code. As a result, SMT solvers ignore implicit semantic input constraints, despite input variables often embodying corresponding semantics. In the example above, the birth year of participants in the dataset should fall within a reasonable range, such as `1900≤birth_year≤2023`; the state of residence should correspond to one of 50 states in the US, such as "CA"; and the height should conform to a distribution, such as `Gaussian(1.70,0.1)`.

Existing SMT solvers are not designed to incorporate such prior knowledge about input variables [18–21]. Therefore, symbolic executors generate concrete inputs without consideration of the likelihood of seeing such solutions in the underlying real-world data distribution. This inherently limits the potential for symbolic executors to generate natural test inputs.

We propose a new symbolic test generation tool, named NaturalSym, which increases test comprehensibility by instilling naturalness constraints during SMT-based input generation. We enhance the naturalness of generated test inputs, while achieving high path coverage as existing symbolic executors. As most DISC applications primarily handle tabular data, NaturalSym incorporates realism constraints for each table column in three formats: (1) the underlying data distribution (e.g., Gaussian or Uniform), (2) the numerical range, and (3) real-world samples from the column. For example, users can specify the preferred value list for `state_of_residence` as `Discrete("CA", "IL", "AZ")` and the range of `birth_year` as `[1900, 2023]`. To generate test inputs, NaturalSym first creates concrete samples for each column based on user annotations. It then prioritizes the use of these realistic values when generating tests for each program path. For the path `birth_year<2005 && state_of_residence!="CA"`, NaturalSym produces `state="IL", birth_year=1990, height=1.75` instead of `state="", birth_year=0, height=0`. Our method leverages both the power of constraint solving and user specifications about the underlying data distribution, producing high-quality tests that achieve superior path coverage and naturalness.

We compare NaturalSym against BigTest [30], a symbolic execution-based DISC test generator that does not model naturalness constraint as we do. Our evaluation shows that NaturalSym achieves the same path coverage as BigTest, yet uncovers 17.4 more injected faults on average. Although both tools consider the same set of program paths, we find that NaturalSym's natural-looking test data better reflects real-world scenarios and is more effective in uncovering program errors than BigTest. As naturalness is a subjective notion without a universally accepted quantitative metric, we utilize a well-known metric *perplexity* [55] from the natural language processing domain as a reasonable proxy to quantify the naturalness of a test. NaturalSym achieves a perplexity score 35.1 points lower than BigTest. In our user studies, participants unanimously preferred the tests generated by NaturalSym, finding that they better aligned with the semantics of the columns. In addition, NaturalSym generates 86.7% fewer numerical outliers compared to BigTest.

We also compare NaturalSym with other test generation tools beyond symbolic execution: (1) NaturalFuzz [36], a coverage-guided fuzzer that aims to improve the naturalness of generated inputs for DISC programs; (2) TestMiner [27], a mining-based test generator; and (3) ChatGPT [3], a representative large language model. NaturalFuzz, TestMiner, and ChatGPT cover 77.5%, 96%, and 77.3% fewer program paths respectively, and find 97.9%, 100%, and 94.8% fewer injected faults than NaturalSym. Our empirical results all suggest that NaturalSym can generate higher quality tests in terms of naturalness, while maintaining high code coverage and bug-finding potential.

The contributions of our work are summarized below:

- We are the first to incorporate the notion of naturalness into symbolic execution-based test generation. NaturalSym does not sacrifice path coverage while only adding a small runtime overhead. In addition, we detected 17.4 more injected faults on average than the state-of-the-art DISC symbolic executor BigTest.
- We enable a developer to specify prior knowledge about the underlying statistical distribution, range constraints, or a dictionary of preferred known values of each column. Additionally, we devise a new algorithm that iteratively invokes SMT solvers to refine the naturalness of the generated test cases. Not all naturalness constraints, specifically statistical distributions, can be encoded as a precondition in SMT constraints easily. Our results show that incorporating statistical distribution reduces numerical outliers by 21.5%.
- We use *perplexity* to quantify naturalness and perform two user studies to assess the quality of test data from users' perspectives, suggesting NaturalSym's advantage over BigTest in terms of naturalness.
- We compare our tool against alternative natural test generators such as NaturalFuzz, TestMiner, and ChatGPT, demonstrating superior path coverage and defect detection. This advantage stems from NaturalSym's unique ability to exploit path constraints via symbolic execution while improving test naturalness.

2 MOTIVATING EXAMPLE

We present a motivating example from our benchmark, Usedcars, to demonstrate the need for realistic input generation. Suppose Alice retrieves the Carvana transaction data and writes a scala-based DISC application to identify the model names and mileage information of used cars that are less than 10 years old and have discounts exceeding \$5,000, as shown in Fig. 1a. She reads two tables from CSV files into Resilient Distributed Datasets (RDDs). She then creates an RDD named `basics` from `input1` using a map operation. The associated UDF ❶ extracts the first column `vehicle_id` and second column `model` from each row. Similarly, `input2` is transformed into `sales`, which

Table 1. Four program paths enumerated by symbolic execution, visualized in Fig. 1c. Valid tests generated by CVC5 are shown for every path condition. In this example, each table only contains a single row.

#	Path condition	input1	input2
φ_1	input1.split(",")(0) = input2.split(",")(1) && input2.split(",")(2).toInt-input2.split(",")(3).toInt>5000 && input2.split(",")(4).substring(0,4).toInt-input2.split(",")(1).toInt<10	","	",0,9000,0,0009,0"
φ_2	input2.split(",")(2).toInt-input2.split(",")(3).toInt≤5000		",,0,0,,"
φ_3	&& input2.split(",")(2).toInt-input2.split(",")(3).toInt>5000 && input2.split(",")(4).substring(0,4).toInt-input2.split(",")(1).toInt≥10		",9,9000,9,9999,"
φ_4	input1.split(",")(0) ≠ input2.split(",")(1) && input2.split(",")(2).toInt-input2.split(",")(3).toInt>5000 && input2.split(",")(4).substring(0,4).toInt-input2.split(",")(1).toInt<10	","	"A,0,9000,0,0009,0"

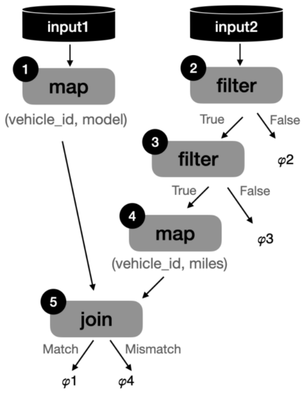
```
val input1 = sc.textFile("car_model.csv")
val input2 = sc.textFile("car_info.csv")
val basics = input1.map❶(row => {
    val vehicle_id = row.split(",")(0)
    val model = row.split(",")(1)
    (vehicle_id, model)
})
val sales = input2.filter❷(row => {
    val price = row.split(",")(2).toInt
    val disc = row.split(",")(3).toInt
    price - disc > 5000
})
.filter❸(row => {
    val sold_year = row.split(",")(4)
        .substring(0,4).toInt
    val pro_year = row.split(",")(1).toInt
    sold_year - pro_year < 10
})
.map❹(row => {
    val vehicle_id = row.split(",")(0)
    val miles = row.split(",")(5).toInt
    (vehicle_id, miles)
})
val result = basics.join❺(sales)
return result
```

(a) usedcars.scala

```
input1: car_model.csv
2404009,Highlander
2161751,Accord
2206151,Tucson

input2: car_info.csv
2404009,2016,37990,31990,2022-07-07,55379
2161751,2010,15880,13990,2022-07-29,101659
2206151,2015,23590,20590,2022-07-20,37331
```

(b) Example input of usedcars.scala



(c) Enumerated program paths in symbolic execution.

Fig. 1. A DISC application, reading Carvana used car transaction records from two tables and summarizing the model and mileage information. An example input is shown in (b) and the graph of program paths is shown in (c).

contains tuples of `(vehicle_id, miles)`. Finally, `result` contains the result of joining `basics` and `sales` on the key `vehicle_id`, yielding model and mileage information for used cars less than 10 years old with discounts exceeding $5,000.

To test the DISC program, Alice runs symbolic execution such as BigTest to thoroughly enumerate program paths and generate corresponding test inputs. Table 1 displays four path constraints, along with tests obtained using SMT solvers such as CVC5. Since the program lacks an aggregation operator like `reduce`, symbolic execution is limited to scenarios where each table contains at most one row. Fig. 3 shows the first path constraint φ_1 in SMT2 format. It represents a non-terminating case where a row in `input2`, corresponding to a used car <10 years old with a discount >$5000, successfully joins with a row from `input1`. For this program path, CVC5 produces a test with a row "," in input1 and a row ",0,9000,0,0009,0" in input2, as visualized in Fig. 2a with its schema.

(a) BigTest generated test case for the path φ_1 in *usedcars.scala*

input1	
vehicle_id	model

input2					
vehicle_id	year	price	sold_price	sold_date	miles
	0	9000	0	0009	0

(b) NaturalSym generated test case for the same path φ_1 in *usedcars.scala*

input1	
vehicle_id	model
2432422	Fit

input2					
vehicle_id	year	price	sold_price	sold_date	miles
2432422	2014	28735	14788	2022-07-05	63342

Fig. 2. (a) and (b) are tests generated by BigTest and NaturalSym for the same path φ_1 shown in Table 1. Both tests are visualized and aligned with the table header.

Symbolic execution can produce high-quality tests in terms of path coverage and bug detection capability, yet the naturalness of the generated test cases is poor since Alice finds test data like ",0,9000,0,0009,0" unrealistic. The last column, indicating a mileage of 0, is unlikely to appear in a used car transaction record. Similarly, "0009" is an invalid date. Alice favors more natural test data, as unrealistic tests put more burden on users to comprehend them. Moreover, bugs triggered by realistic data are more worthy of fixing than bugs induced by unrealistic, unlikely data. Hence, the naturalness of tests is crucial from both perspectives.

To enhance test naturalness, NaturalSym allows users to provide semantic information about inputs, such their the underlying statistical distributions. In the running example, Alice can label the `sold_price` column as `Gaussian(`$\mu = 2 \times 10^4$`,` $\sigma = 10^4$`)` and the `model` column as Discrete("Ford", "Hyundai"). Rather than solely invoking CVC5 with path condition φ_1, NaturalSym will draw concrete samples for each column based on user annotations and prioritize the use of these realistic values in constraint solving. For instance, NaturalSym generates a realistic test, as visualized in Fig. 2b. The `sold_price` is set as 14788 rather than 0; the car model is "Ford" instead of an empty string; and the `sold_date` is "2022-07-05" rather than "0009." All elements in the generated test appear more natural-looking while satisfying the path condition φ_1.

3 METHODOLOGY

In this section, we formalize the problem of improving naturalness in test generation as an optimization problem, known as an OMT problem (the optimization version of an SMT problem) [48]. We briefly discuss the challenges of setting up reasonable naturalness oracles and solving optimization problems. Then, we demonstrate how users can specify their prior knowledge about underlying distributions, numerical ranges, and preferred values over each input variable in NaturalSym. Finally, we propose an effective greedy algorithm to improve test naturalness, which introduces 1.47× runtime overhead compared to plain symbolic execution.

3.1 Formalization

Symbolic execution enumerates program paths and collects the logical constraints along each path, the so-called path conditions. Table 1 lists four path conditions of the motivating program in Fig. 1a. A path condition is usually expressed in SMT-LIBv2 format [15], also known as a Satisfiability Modulo Theories (SMT) formula. For example, Fig. 3 shows a concrete path condition φ_1 from Table 1 written in SMT-LIBv2 format. A world ω is a function that maps a variable to a concrete value, *i.e.,* a complete variable assignment. The set of worlds that satisfy the path condition φ forms the models of φ, *i.e.,* $Mods(\varphi) = \{\omega \mid \omega \vDash \varphi\}$. Alternatively speaking, $Mods(\varphi)$ includes all concrete assignments that satisfy the path condition φ. To generate a test that executes the path φ_1, traditional symbolic executors will send the path condition to SMT solvers such as Z3 [25] or CVC5

```
(assert (= input2 (str.++ (str.++ (str.++ (str.++ (str.++ input2_d0 (str.++ ","
    input2_d1)) (str.++ "," input2_d2)) (str.++ "," input2_d3)) (str.++ "," input2_d4
    )) (str.++ "," input2_d5))))
(assert (= input1 (str.++ input1_d0 (str.++ "," input1_d1))))
(assert (>= (str.len input2_d4) 4))
(assert (= input1_d0 input2_d0 )))
(assert (and (> (- (str.to_int input2_d2) (str.to_int input2_d3)) 5000) (and (
    isinteger input2_d3) (isinteger input2_d20))))
(assert (< (- (str.to_int (str.substr input2_d4 0 (- 4 0))) (str.to_int input2_d1) )
    10) (and (isinteger input2_d1) (and (isinteger (str.substr input2_d4 0 (- 4 0)))
    (and (isinteger input2_d5)))))
```

Fig. 3. The actual path condition φ_1 for the first path in Table 1, written in SMT2 format. φ_1 corresponds to the case where both tables contain a row with the same *vehicle_id*, while the associated used car is <10 years old with a discount >$5,000.

[14] to obtain a valid solution. However, SMT solvers will return an arbitrary solution within the solution space $Mods(\varphi)$ without any regard for naturalness.

Therefore, we assume a naturalness oracle exists for any input assignments, $nat\_score(\omega)$. The naturalness oracle quantifies the degree of naturalness as a numerical score. To improve naturalness, we not only desire solvers to provide a valid solution but also to maximize the naturalness score, which turns out to be an Optimization Modulo Theories (OMT) problem [48].

$$\arg\max_{\omega}\{nat\_score(\omega) \mid \omega \vdash \varphi\}$$

However, this poses a challenging problem on two fronts. (1) Since naturalness is inherently subjective, defining an oracle to quantify its degree is difficult. In addition, the natural characteristics of tabular data, including string format restrictions, numerical range, and foreign key relationships connecting multiple tables, are diverse and complex. (2) OMT solving is computationally more demanding than standard SMT solving. Thus, it is necessary to devise an appropriate naturalness oracle that encapsulates key aspects of naturalness and to design an efficient algorithm that adds little runtime overhead compared to standard SMT solving. We present our solution to these challenges in the next sections.

3.2 Additional Prior Knowledge Annotation

Existing symbolic executors collect only path constraints. To enhance test naturalness, NaturalSym enables users to specify their prior knowledge in an additional configuration file. Specifically, for each column, users can (1) specify the underlying distribution, (2) set an optional value range for numbers, or (3) provide some concrete examples from actual data. Currently, NaturalSym supports Gaussian distributions, uniform distributions over intervals, and uniform discrete distributions, with potential for future extensions. The DSL of our annotation language is shown in Fig. 4.

An example of user-specified knowledge for the used car transaction record table is displayed in Fig. 5, where annotations are aligned with each column header. The `miles` column is expected

$$\begin{aligned}
Schema:&\quad \mathcal{R}^{+}\\
\mathcal{R}:&\quad \mathcal{V} := \mathcal{D}(,\mathcal{D})^{*}\\
\mathcal{D}:&\quad (l \leq)^{?} Gaussian(\mu,\sigma)(\leq r)^{?} \mid Discrete(\{d_1, d_2, d_3, \cdots, d_k\}) \mid Uniform(l, r)
\end{aligned}$$

Fig. 4. DSL for user annotation, showing how developers specify semantics for each table column.

input1	
vehicle_id	model
Discrete(2432422,...)	Discrete("Fit",...)

input2					
vehicle_id	year	price	sold_price	sold_date	miles
Discrete(3432422,...)	Uniform(2009,2022)	Gaussian(3×10^4,10^4)	Gaussian(2×10^4,10^4)	Discrete("2022-07-03",...)	100≤Gaussian(50000,10^4)

Fig. 5. User annotations for the input tables of the used car program, as shown in Fig. 1a.

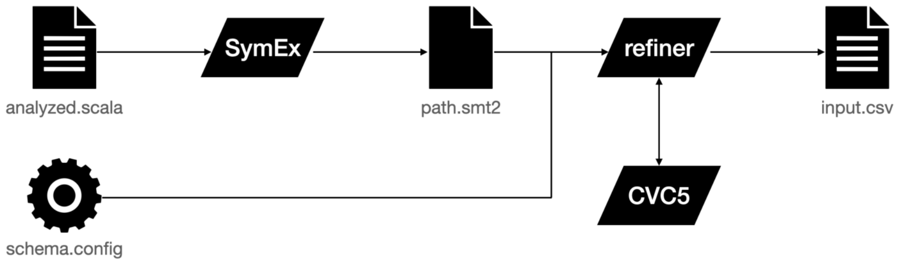


Fig. 6. Pipeline of NaturalSym. In addition to the target program, NaturalSym reads a configuration file in which users specify prior knowledge about inputs. Then, the refiner combines realistic value choices from the configuration file and the path conditions from the symbolic executor to invoke the SMT solver iteratively.

to follow a Gaussian distribution with parameters $\mu = 50000$ and $\sigma = 10^4$, and a minimum mileage of 100, annotated as `100 ≤ Gaussian(50000,10`^{4}`)`. Similarly, the `model` column is labeled with `Discrete("Fit",...)`, indicating it should match actual car model names. Notably, user annotations can be incomplete. For example, the `model` column may be labeled simply with a preferred subset such as `Discrete("Ford", "Hyundai")`, without listing all available car models on the market.

While NaturalSym has the potential to support a wider variety of distribution types and incorporate more types of prior knowledge in the future, such as consistency constraints across columns and tables, empirical experiments show that the currently supported annotations effectively capture sufficient prior knowledge. They also help mitigate several common issues with unnatural data, such as numerical outliers and missing values.

3.3 Our Algorithm

Fig. 6 illustrates the pipeline of NaturalSym. Initially, NaturalSym reads user annotations from an additional configuration file and collects path conditions through symbolic execution on the target program. For each program path, it then enhances test naturalness by iteratively invoking the SMT solver. In this section, we detail how user prior knowledge is transformed into the naturalness oracle and how we maximize the naturalness score of the generated tests.

Oracle selection. Based on user prior knowledge, we draw M_0 concrete samples S_{ivar} for each input variable *ivar*, which corresponds to a table element. For example, if a column containing *ivar* is annotated to follow a Gaussian distribution $Gaussian(5, 2)$, we directly sample M_0 concrete values using a Gaussian sampler. Then, we define the naturalness oracle as follows:

$$nat\_score(\omega) = \sum_{ivar \in I} [\omega(ivar) \in S_{ivar}] \,. \tag{1}$$

The naturalness score of a model ω is determined by the number of input variables whose values fall within their corresponding sample sets S_{ivar}. Consider a path condition `date.substring(0,4) == year`, with the sample sets S_{date}={"2018-01-01","2017-01-01"} and S_{year}={"2017", 2015"}. In this scenario, the solution `date`="2017-01-01" and `year`="2017" achieves a naturalness score of 2, as both variables appear in their respective sample sets. This solution is optimal under the defined naturalness oracle. In general, our naturalness score guides the solution towards selecting values from the sample sets. Given that these sample sets are derived from user annotations, our optimization objective, namely the naturalness score, aligns with our goal of enhancing test naturalness.

Greedy optimization solving. Considering the inefficiency of OMT solving, NaturalSym uses a more efficient greedy algorithm to generate realistic inputs. Our algorithm, shown in Algo. 1, takes

Algorithm 1: Greedy algorithm to improve naturalness

```
Input: A program path condition $\varphi$, an input variable set $I$, and user specified knowledge $\mathcal{K}$.
Output: A valid and natural solution $Sol$
1  $Result \leftarrow constraint\_solver(\varphi)$
2  if $Result$ is UNSAT then
3      return None
4  end
5  $Sol \leftarrow$ returned model in $Result$
6  for $ivar \in I$ do
7      $S_{ivar} \leftarrow M_0$ concrete values based on $\mathcal{K}$
8      if $ivar$ is a free variable in $\varphi$ then
9          $v \leftarrow$ arbitrary value in $S_{ivar}$
10         $Sol(ivar) \leftarrow v$
11         $\varphi \leftarrow \varphi \wedge (ivar = v)$
12     end
13     else
14         $\varphi' \leftarrow \varphi \wedge (ivar \in S_{ivar})$
15         $Result' \leftarrow constraint\_solver(\varphi')$
16         if $Result'$ is SAT then
17             $Sol \leftarrow$ returned model in $Result'$
18             $\varphi \leftarrow \varphi'$
19         end
20     end
21 end
22 return $Sol$
```

a path condition φ and user knowledge $\mathcal{K}$ as input. We first invoke the SMT solver once to obtain an initial solution for the specific path. Then, our algorithm works in a step-by-step manner to improve test naturalness.

In each step, we try to improve the naturalness of a new input variable *ivar*, where each input variable represents a table element. Based on user annotation $\mathcal{K}$, we draw M_0 concrete realistic instances S_{ivar} for *ivar*. If *ivar* is not part of the path condition, we can freely assign any value from its sample set S_{ivar} without making additional SMT calls. Otherwise, we attempt to restrict its value choices to those within S_{ivar}. Specifically, we conjoin the current path condition φ with $ivar \in S_{ivar}$ and invoke the SMT solver. The new constraint of value choices can be expressed as `assert (or (= ivar` $choice_1$`) ... (= ivar` $choice_k$`))` in SMT2Lib format. If the modified path condition is satisfiable, we achieve a refined solution and increase the naturalness score by 1. We greedily retain the value choice constraints on *ivar* and update our solution. If unsatisfactory, we discard the value choice constraint for *ivar* and revert to the previous path condition.

Fig. 7 illustrates our algorithm using a concrete example. Initially, NaturalSym receives a path condition from the underlying symbolic executor and obtains a primitive solution using CVC5. In step 1, we sample two instances {2432422,2432244} for `vehicle_id` based on the user annotation $Discrete(2432422, \ldots)$. We attempt to constrain the value choices of `input1.vehicle_id` by conjoining the current path condition with `input1.vehicle_id ∈ {2432422,2432244}`. Upon invoking CVC5, we obtain a refined solution where `vehicle_id` is set as 2432422. In step 2, recognizing that `model` is a free variable, we directly assign it the concrete value "Fit" from its sample set. The path condition is updated accordingly to retain this choice in subsequent steps.

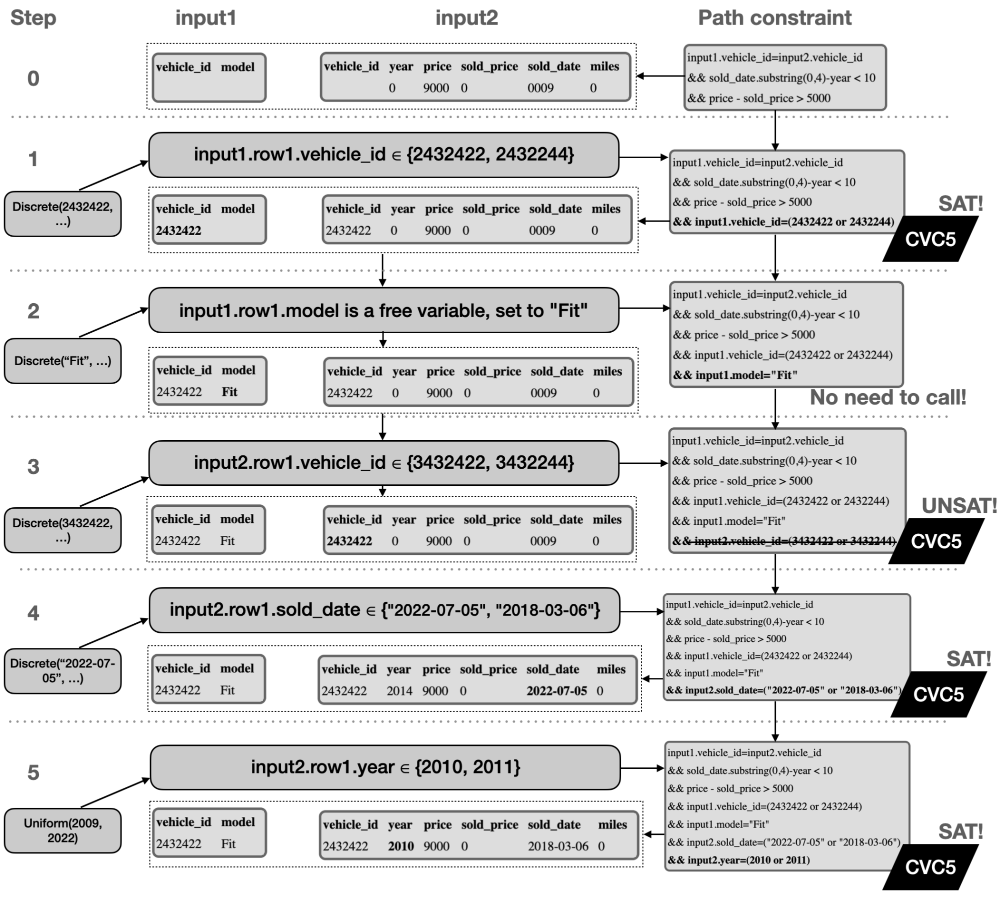

Step
input1
input2
Path constraint
0
vehicle_id model
vehicle_id year price sold_price sold_date miles
0 9000 0 0009 0
input1.vehicle_id=input2.vehicle_id
&& sold_date.substring(0,4)-year < 10
&& price - sold_price > 5000
1
input1.row1.vehicle_id ∈ {2432422, 2432244}
Discrete(2432422, ...)
2432422
2432422 0 9000 0 0009 0
&& input1.vehicle_id=(2432422 or 2432244)
SAT!
CVC5
2
input1.row1.model is a free variable, set to "Fit"
Discrete("Fit", ...)
2432422 Fit
&& input1.model="Fit"
No need to call!
3
input2.row1.vehicle_id ∈ {3432422, 3432244}
Discrete(3432422, ...)
&& ~~input2.vehicle_id=(3432422 or 3432244)~~
UNSAT!
4
input2.row1.sold_date ∈ {"2022-07-05", "2018-03-06"}
Discrete("2022-07-05", ...)
2432422 2014 9000 0 2022-07-05 0
&& input2.sold_date=("2022-07-05" or "2018-03-06")
5
input2.row1.year ∈ {2010, 2011}
Uniform(2009, 2022)
2432422 2010 9000 0 2018-03-06 0
&& input2.year=(2010 or 2011)

Fig. 7. A running example of our algorithm, refining the test for the program path φ_1 (Table 1) of *usedcars.scala* (Fig. 1a). φ_1 represents the scenario where both tables contain a row with the same vehicle_id, while the corresponding used car has an age <10 and a discount >5000. In each iteration, NaturalSym dynamically modifies the path condition by adding new naturalness constraints. For simplicity, the path condition is not written in strict SMT format, and M_0 is set to 2.

In step 3, constraining `input2.vehicle_id` to its sample set {3432422,3432244} is impossible due to a conflict with the condition `input1.vehicle_id=input2.vehicle_id`. Hence, we remove the constraint on `input2.vehicle_id` and revert the path condition to its previous state. Importantly, even if we successfully constrain a variable *ivar* within its sample set, we still keep various options for *ivar* in the path condition rather than immediately fixing its value. For example, setting `sold_date` to "2022-07-05" in step 4 would conflict with potential `year` choices {2010, 2011} in step 5 because the car is expected to be less than 10 years old, as defined by the condition `sold_date.substring(0,4).toInt - year < 10`. Therefore, we keep various choices {"2018-03-06", "2022-07-05"} for `sold_date`, enhancing the potential for further refinement.

In general, our algorithm obtains realistic concrete instances for each input variable based on user-specified knowledge and refines these variables greedily. Even if all refinement attempts fail, the algorithm still yields a valid solution for any satisfiable path condition. We have further optimized the handling of free variables to reduce the additional SMT calls, which are the primary bottleneck of the algorithm.

Table 2. DISC programs and their descriptions.

Program	Description	#Datasets
Airport (P1)	Filter all the open airports with an elevation above 1000 feet in the region code 'CA', and output their GPS coordinates information.	2
Credit (P2)	Compute the maximum installment payment amount made by individuals in each profession for purchasing a new car.	1
Usedcars (P3)	Analyze the Carvana transaction records to identify the make and mileage information of used cars with discounts greater than 5000 dollars and less than 10 years old.	2
Movie (P4)	Calculate the number of old films in the IMDB database released bewteen 1900 to 1960, with audience ratings of no less than 4 points, categorized by genre.	1
Transit (P5)	Calculate the total layover time of passengers at each airport.	1
TPC-DS Q1	Find customers who have returned items more than 10,000 us dollors for a store in a given state for a given year.	4
TPC-DS Q3	Report the total extended sales price per item brand of a specific manufacturer for all sales in a specific month of the year.	3
TPC-DS Q6	List all the states and the number of customers who during a given month bought items with the price tag higher than 100.	5
TPC-DS Q7	Compute the total quantity of promotional items sold in stores where the promotion is not offered by mail or a special event. Restrict the results to a specific gender, marital, and educational status.	5
TPC-DS Q12	For each item in a list of given categories, during a 30-day time period, sold through the web channel, compute the sales of that item.	3
TPC-DS Q15	Report the total catalog sales for customers in selected geographical regions or who made large purchases for a given year and quarter.	4
TPC-DS Q19	Compute the total revenue by products bought by out of zip code customers for a given year, month, and manager.	6
TPC-DS Q20	Compute the total revenue by item class and total revenue for specified item categories and time periods.	3

4 EVALUATION

Our evaluation aims to answer the following research questions.

- RQ1: Does NaturalSym generate more realistic inputs than plain symbolic execution?
- RQ2: Does NaturalSym achieve high path coverage and fault detection capability?
- RQ3: How much overhead is incurred by refining test naturalness in NaturalSym?
- RQ4: How much does the knowledge of statistical distributions contribute to NaturalSym?
- RQ5: Does NaturalSym perform better than alternative other test generators beyond symbolic execution?

In Section 4.1, we use perplexity scores from large language models as a naturalness proxy to compare NaturalSym against the state-of-the-art symbolic executor BigTest. In Section 4.2, we apply mutation testing [37] to measure the fault detection capability of NaturalSym and BigTest and report the runtime overhead introduced by our tool. Due to the lack of real-world DISC faulty benchmarks, we inject faults into our benchmarks similar to previous works [30, 36]. A test generator detects a fault when at least one test yields different outputs between the original and faulty program. In Sections 4.3 and 4.4, we design two user study tasks to assess naturalness quality. In Section 4.5, we compute the portion of numerical outliers in the generated values from Gaussian columns and conduct an ablation study by ignoring statistical distributions in user annotations. In Sections 4.6, 4.7, and 4.8, we compare NaturalSym against othe natural test generators.

Baselines. We compare NaturalSym against (1) BigTest [30], a symbolic execution-based DISC testing tool. In addition, we compare NaturalSym against alternative natural test generators: (2) NaturalFuzz [36], a coverage-guided fuzzer that extracts column values from real datasets and then weaves them together to improve both test naturalness and statement coverage; (3) TestMiner

Table 3. The median perplexity across 18 input tables. Lower perplexity suggests better naturalness.

Original	BigTest	NaturalSym	NaturalFuzz	NaturalFuzz-H
2.27	40.01	4.90	3.48	3.67
NaturalFuzz-P	TestMiner	TestMiner-DM+	ChatGPT	
79.35	4.73	36.36	16.92	

[27], which predicts suitable input values based on mined tests from Maven Central Repository [7]; (4) ChatGPT [3], which is a representative large language model. We further compare our tool against the variants of NaturalFuzz and TestMiner: (5) NaturalFuzz-H (NaturalFuzz-Hybrid), taking seeds from both the original dataset and BigTest; (6) NaturalFuzz-P (NaturalFuzz-PureSymbolic), taking seeds from BigTest only; (7) TestMiner-DM+ (TestMiner-DataMining+), with extra data-mining from DISC program datasets.

Benchmarks. Our benchmark suite originates from two sources: five subject programs similar to prior Big Data Analytics debugging or testing and another eight subject programs derived from the *TPC-DS* benchmark [45], a commercial benchmark suite. Transit is developed and used by [30, 31]. Usedcars, Airport, Credit, and Movie are custom Apache Spark applications based on online publicly available datasets [1, 2, 6, 32]. The *TPC-DS* benchmark contains 99 real-word SQL queries along with vital business information, such as customer, order, and product data. The TPC-DS benchmark was originally designed as a standard for performance testing of operating systems and databases. NaturalFuzz [36] translated eight SQL queries into Scala-based Apache Spark applications to evaluate code coverage and bug-finding capabilities of fuzzing tools. We further refactor floating point variables to integers as NaturalSym's symbolic executor does not currently support floating point modeling. Table 2 shows the detailed descriptions of benchmark programs. In total, our benchmark suite contains 13 programs.

Fault Injection. To test the fault detection capability of NaturalSym, we adopt mutation strategies from previous works to inject faults into each program. BigTest [30] investigated seven common types of errors in DISC applications, such as *using the wrong join type*. We manually insert each type of error into each original program if applicable. For example, we add into our benchmark suite a faulty program with `a.rightOuterJoin(b)` when there exists `a.join(b)` in a subject application. NaturalFuzz [36] replaced each binary operator in DISC programs with a random wrong operator to extend its faulty program suite. For example, `a.reduceByKey(_+_)` may be mutated as `a.reduceByKey(_-_)`. We use the same script from [36] to generate faulty mutants for the 8 *TPC-DS* programs. In total, our benchmark suite contains 80 faulty programs.

Implementation. NaturalSym uses BigTest to collect path constraints from subject programs. Scala programs are compiled into Java bytecode and are then analyzed by Symbolic Path Finder [52]. We refer our readers to BigTest [30] for more details. NaturalSym refines concrete choices based on user specifications through sampling and then iteratively invokes SMT solvers to solve the path constraints conjunct with value choice constraints. In evaluation, we use the same symbolic executor and backend constraint solver, CVC5, for BigTest and NaturalSym to ensure a fair comparison.

Experiment Setup. We set the size of sample sets as $M_0 = 10$, as ten concrete examples provide sufficient realistic choices and acceptable overhead empirically. In user studies, we recruit nine computer science PhD students with extensive coding experience via email. We run our experiments on Apache Spark 2.4 and HDFS 1.0.3. All the experiments are conducted on a Dell PowerEdge R630 Server with 224GB RAM and 2 Intel Xeon E5-2640 v3 2.60GHz 8-core processor CPUs running Ubuntu 22.04.

Table 4. Number of covered program paths by each tool averaged over 5 runs. "-" stands for crashes on specific baselines. NaturalSym preserves the same path coverage as BigTest and surpasses all other baselines.

	Subject program													
	P1	P2	P3	P4	P5	Q1	Q3	Q6	Q7	Q12	Q15	Q19	Q20	Total
NaturalSym	5	5	4	4	6	7	5	8	11	11	23	10	11	110
BigTest	5	5	4	4	6	7	5	8	11	11	23	10	11	110
NaturalFuzz	2	-	1	-	-	1.8	2	5	3.2	2.6	3.2	2	2	24.8
NaturalFuzz-H	2	-	1.8	-	-	-	2.2	-	2.4	2	3.8	2	-	16.2
NaturalFuzz-P	2	-	2.8	-	-	2	2.2	2.4	4.6	3	3.8	2	1.4	26.2
TestMiner	1	0	1	1	0	0	0	0	0	0.2	0.8	0	0.4	4.4
TestMiner-DM+	2.6	1	1	2	1	1	2	2.8	3	2.4	3.6	2.4	2.6	27.4
ChatGPT	1	1	1	1	1	1	2	4	3	3	2	2	3	25

Table 5. Number of seeded faults detected by NaturalSym against baselines averaged over 5 runs. "-" stands for crashes on specific baselines. NaturalSym detects 76.4 injected faults out of 80.

	Subject program													
	P1	P2	P3	P4	P5	Q1	Q3	Q6	Q7	Q12	Q15	Q19	Q20	Total
Seeded faults	5	5	5	5	6	7	9	6	7	5	7	6	7	80
NaturalSym	5	5	5	5	5.4	5	9	5	7	5	7	6	7	76.4
BigTest	5	5	4	5	6	4	4	4	4	3	7	4	4	59
NaturalFuzz	0	-	0	-	-	0.4	0.2	0	0	0	1	0	0	1.6
NaturalFuzz-H	0	-	3.2	-	-	-	0.2	-	0	0	0.8	0	-	4.2
NaturalFuzz-P	0.6	-	3.2	-	-	1.6	0.6	2	1	0	6	0	0	15
TestMiner	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TestMiner-DM+	0	1	0	0	0	0	0	0	0	0	0	0	0	1
ChatGPT	1	1	1	1	0	0	0	0	0	0	0	0	0	4

4.1 Perplexity Measurement

We quantify the naturalness of generated tests using a well-known metric called *perplexity* [55]. For each input table, we fine-tune Distilled-GPT2 [4] on its original dataset, where each row is converted into plain text delimited by commas. We then evaluate the perplexity of rows generated by NaturalSym and baselines using the fine-tuned models. Although perplexity is widely adopted to assess text quality, it fails to accurately reflect the unnaturalness of ",,,,", an empty row in tabular data, which reports a low perplexity score due to its highly repetitive and common pattern. To address this limitation, we insert a *<PAD>* token in each empty column to penalize tools for generating such tests with numerous empty columns.

Table 3 shows the median perplexity scores for each baseline along with the original data. We observe that the tests generated by NaturalSym are more natural than those generated by BigTest, as it achieves a perplexity that is 35.1 points lower. The only input dataset of Movie (P4) contains 5 columns: `MovieName,Year,Genre,Ratings,Duration`. For the path condition `1960≤year`, BigTest generates a row ",9000,,," which sets the year to 9000 while leaving other parts completely empty. In contrast, NaturalSym generates a row "Minions: The Rise of Gru,1991,Sci-Fi,5,02:11". NaturalSym leverages user-specified knowledge, making its generated tests align well with the original data schema.

4.2 Fault Detection Capability

Path coverage. Table 4 shows the number of program paths covered by NaturalSym and baselines. It is observable that NaturalSym and BigTest achieve identical path coverage on each benchmark. This is because NaturalSym enumerates the same set of program paths as BigTest. Furthermore, NaturalSym adopts an iterative approach to improve test naturalness. If NaturalSym finds that

```
val result = input1.map {s =>
  def getDiff(arr: String, dep: String): Int = {
    val a_min = Integer.parseInt(arr.substring(3, 5))
    val a_hr = Integer.parseInt(arr.substring(0, 2))
    val d_min = Integer.parseInt(dep.substring(3, 5))
    val d_hr = Integer.parseInt(dep.substring(0, 2))
    val arr_min = a_hr * 60 + a_min
    val dep_min = d_hr * 60 + d_min
    if ( dep_min - arr_min < 10 ) {
        //should be dep_min - arr_min < 0
      return 24 * 60 + dep_min - arr_min
    }
    return dep_min - arr_min
  }
  val tokens = s.split(",")
  val arrival_hr = tokens(2).substring(0, 2)
  val diff = getDiff(tokens(2), tokens(3))
  val airport = tokens(4)
  (airport + arrival_hr, diff)
}.filter { v => v._2 < 45 }
.reduceByKey(_ + _)
.map(m => m._1 +","+m._2)
return result
```

(a) transitWrongPredicate.scala

```
arr = input1.split(",")(2)
&& dep = input1.split(",")(3)
&& arr_min = arr.substring(0,2).toInt * 60 + arr.substring
      (3,5).toInt
&& dep_min = dep.substring(0,2).toInt * 60 + dep.substring
      (3,5).toInt
&& diff = dep_min - arr_min
&& diff >= 0
&& diff < 45
```

(b) A non-terminating path in the original program *transit.scala*.

passenger_id	transit_date	arrival_time	departure_time	transit_airport
		00A00	00C00	

(c) Generated by BigTest.

passenger_id	transit_date	arrival_time	departure_time	transit_airport
P857	2018-10-04	08:08	08:20	LAX

(d) Generated by NaturalSym.

Fig. 8. In (a) transitWrongPredicate.scala, within the UDF of the first map operation, the <0 condition is mistakenly written as <10. (b) shows a terminating path in transit.scala. (c) and (d) are concrete tests for the given path in (b), generated by BigTest and NaturalSym respectively.

constraining the value choices of a variable proves unsuccessful, it will revert to the solution obtained in the previous round. Therefore, NaturalSym can always produce valid test inputs for every feasible path, thus maintaining comprehensive path coverage while enhancing naturalness.

Fault detection. Table 5 shows the number of injected bugs found by NaturalSym and other tools. NaturalSym finds 17.4 more injected bugs than BigTest on average. Among them, there is one bug that BigTest can find but NaturalSym cannot, as illustrated in Fig. 8. Conversely, there are 18 bugs that NaturalSym can detect but BigTest cannot, such as shown in Fig. 9.

Fig. 8a shows the only injected bug that BigTest can find but NaturalSym cannot. This application reads transit records from an input table and then calculates the transfer duration for each record. The highlighted if-statement corresponds to the logic for handling transfers that span across days. Specifically, a day is added when the transfer duration is negative. However, the condition in the faulty program is wrongly written as `dep_min - arr_min < 10`. BigTest generates a test where the transfer duration is 0, which causes the faulty program to add one day to the transfer duration erroneously. In contrast, NaturalSym generates more realistic data but misses this bug. This is because the faulty condition is only triggered when the difference falls within the range [0,10), but NaturalSym sets `arrival_time` to "08:08" and `departure_time` to "08:20".

Even though "00A00" and "00C00" look unrealistic, the test generated by BigTest can detect this bug. This illustrates that greater test naturalness does not necessarily correlate with superior bug-finding potential. However, in our empirical evaluation, the scenario where NaturalSym missed bugs that BigTest detected occurred only once. Since NaturalSym incrementally refines test naturalness, it has the capacity to retain intermediate tests generated in each round. As the tests generated in its initial round are identical to those generated by BigTest without naturalness consideration, NaturalSym's generated tests can fully encompass those of BigTest. In conclusion, refining test naturalness does not undermine NaturalSym's ability to detect bugs.

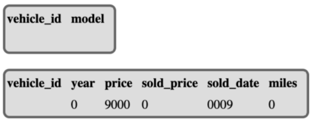
Fig. 9a illustrates a faulty program where NaturalSym detects a bug that BigTest does not. The key and value within the first map operation are incorrectly swapped, as highlighted in red. This program is designed to collect information on used cars less than 10 years old with discounts exceeding $5000. Each row in `basics` contains a tuple `(vehicle_id, model)`, and each row in

```scala
val basics = input1.map(row => {
    val vehicle_id = row.split(",")(0)
    val model = row.split(",")(1)
     (model, vehicle_id)
    // should be (vehicle_id, model)
})
val sales = input2.filter(row => {
    val price = row.split(",")(2).toInt
    val disc = row.split(",")(3).toInt
    price - disc > 5000
})
.filter(row => {
    val sold_year = row.split(",")(4)
        .substring(0,4).toInt
    val pro_year = row.split(",")(1).toInt
    sold_year - pro_year < 10
})
.map(row => {
    val vehicle_id = row.split(",")(0)
    val miles = row.split(",")(5).toInt
    (vehicle_id, miles)
})
val result = basics.join(sales)
.map(row => row._2._1 + "," + row._2._2)
return result
```

(a) usedcarsWrongKV.scala

```
input1.split(",")(0) = input2.split(",")(0)
&& discount = input2.split(",")(2).toInt - input2.split(",")(3).
       toInt
&& age = input2.split(",")(4).substrig(0,4).toInt - input2.split(
       ",")(1).toInt
&& discount > 5000
&& age < 10
```

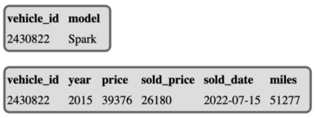
(b) A non-terminating path in the original program *usedcars.scala*.



vehicle_id	model

vehicle_id	year	price	sold_price	sold_date	miles
	0	9000	0	0009	0

(c) Generated by BigTest.



vehicle_id	model
2430822	Spark

vehicle_id	year	price	sold_price	sold_date	miles
2430822	2015	39376	26180	2022-07-15	51277

(d) Generated by NaturalSym.

Fig. 9. In (a) usedcarsWrongKV.scala, the key and value of the returned tuple in the first map operation are reversed. (b) shows a non-terminating path in usedcars.scala. (c) and (d) are concrete tests for the given path in (b), generated by BigTest and NaturalSym respectively.

sales contains a tuple (vehicle_id, miles). The design is for basics and sales to join on vehicle_id. However, the faulty program erroneously tries to match the model from input1 with the vehicle_id from input2 during the join operation.

Fig. 9b represents a scenario involving a used car with age<10 and discount>5000. In the test generated by BigTest, both the original and the faulty programs output a row ",0". This occurs because the path condition requires only that the vehicle_id from both input1 and input2 match, without defining the relationship between vehicle_id and model. Although vehicle_id and model are unlikely to be identical in practice, they are indistinguishable in the BigTest generated test, thereby obscuring the injected fault. In contrast, NaturalSym produces more realistic values, implicitly leveraging the real-world distinction that vehicle_id and model should differ. Consequently, the original program outputs a row "Spark,51277" for the test generated by NaturalSym, whereas the faulty program yields no output. With improved naturalness, NaturalSym successfully detects this injected semantic bug.

Another common scenario where NaturalSym detects more injected faults occurs when a program incorrectly aggregates a column using reduce((a,b)=>a-b) instead of reduce((a,b) => a+b). If the aggregated column affects no branching predicate, CVC5 defaults it to zero. Consequently, the final result will remain zero, regardless of whether the operator is mistakenly written as subtraction or multiplication. Therefore, such faults cannot be detected using tests generated by BigTest. In contrast, NaturalSym can detect such errors because it produces realistic values rather than defaulting to zero. In summary, NaturalSym is more effective at detecting semantic faults compared to BigTest, despite its primary focus on enhancing test naturalness.

Runtime overhead. Fig. 10 shows the runtime comparison between BigTest and NaturalSym across 13 benchmarks. NaturalSym optimizes performance by avoiding unnecessary SMT calls for unconstrained variables. On average, NaturalSym spends 1.47× more time than BigTest. Specifically, in the most time-consuming benchmark Transit (P5), NaturalSym's runtime is 2.22× that of BigTest. Excluding the time for the symbolic executor to collect path conditions and verify

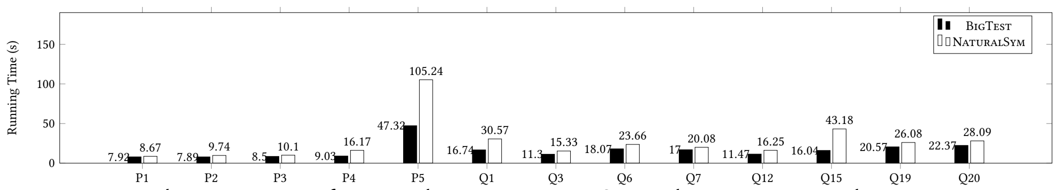
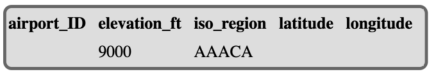
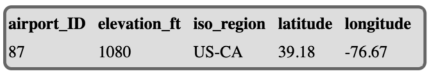


Fig. 10. The comparison of runtime between NaturalSym and BigTest averaged over 5 runs.



airport_ID	elevation_ft	iso_region	latitude	longitude
	9000	AAACA		

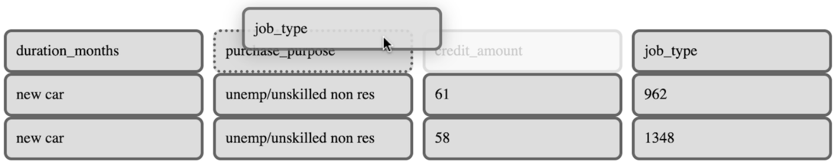
(a) Generated by BigTest.



airport_ID	elevation_ft	iso_region	latitude	longitude
87	1080	US-CA	39.18	-76.67

(b) Generated by NaturalSym.

Fig. 11. Example table generated by BigTest and NaturalSym.



duration_months	purchase_purpose	credit_amount	job_type
new car	unemp/unskilled non res	61	962
new car	unemp/unskilled non res	58	1348

Fig. 12. Interface for user study 1. Column names are randomly shuffled and users can drag them to match the appropriate data columns or mark them as unmatchable.

unsatisfiable path constraints, our iterative SMT solving process takes 9.64× longer time than standard SMT solving on average.

4.3 User Study 1: Column Name Identification

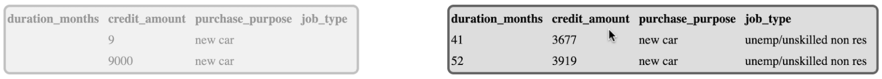
In user study 1, we investigate participants' ability to deduce the appropriate column names based solely on table content. Ideally, the content of each table column should clearly reflect the semantics associated with its column name. If users can accurately match the majority of column names, it suggests that the table is realistically generated. For example, in the table produced by BigTest as shown in Fig. 11, the value under `iso_region` is "AAACA", which does not resemble a typical ISO code. Without knowledge of the `airport_ID` format, users might wrongly regard "AAACA" as an `airport_ID`. In contrast, NaturalSym generates more recognizable values like "US-CA", aiding in the correct identification of column names.

To conduct this user study, we collect all tests generated by both BigTest and NaturalSym for the five custom programs (P1-P5). As shown in Fig. 12, we design a user study interface where each box represents a table element. We randomly shuffle both the headers and the content of each table, starting with all column names initially mismatched with their actual content. Through this interface, users can drag a column name to the corresponding data column or click to disregard unmatched names. After users complete their selections, we tally the numbers of disregarded, incorrectly matched, and correctly matched column names. We exclude this experiment from the *TPC-DS* benchmarks for two main reasons: (1) the input tables in *TPC-DS* programs have an average of 22.6 columns, complicating user interaction; (2) massive numeric columns in *TPC-DS* programs are difficult to distinguish, making the column guessing task pointless.

We recruited nine PhD students with substantial coding experience for this user study. The results are presented in Table 6. Generally, users correctly identified column names in tests form NaturalSym 7.26× more than in those from BigTest. For the data generated by BigTest, 65% of the column names were marked as unmatchable, primarily due to BigTest producing a significant amount of empty values. Even among the column names that users attempted to match, only 29.5% were matched correctly. In contrast, users successfully deduced 76.1% of the column names from the

Table 6. Comparison of user study results between BigTest and NaturalSym. For user study 1, #disregarded indicates the number of column names that users find hard to match with data columns; #mismatched and #matched represent the number of column names that users match incorrectly and correctly. For user study 2, #marked indicates the number of generated tables that users find as realistic.

Program	Table	User Study 1: Column Guessing						User Study 2: User Preference	
		Generated by BigTest			Generated by NaturalSym			by BigTest	by NaturalSym
		#disregarded	#mismatched	#matched	#disregarded	#mismatched	#matched	#marked	#marked
P1	input1	12.3	4.1	3.6	0.4	7.0	12.6	0.0	4.0
P1	input2	4.2	0.8	1.0	0.6	0.4	5.0	0.2	2.8
P2	input1	11.8	4.1	4.1	1.0	1.7	17.3	0.0	4.6
P3	input1	3.6	0.4	0.0	0.0	0.2	3.8	0.0	2.0
P3	input2	14.8	8.3	0.9	0.0	4.4	19.6	0.0	4.0
P4	input1	13.6	3.1	3.3	0.9	0.1	19.0	0.0	3.8
P5	input1	19.8	10.1	0.1	4.9	7.9	17.2	0.0	4.3
Total		80.0	31.0	13.0	7.8	21.8	94.4	0.2	25.4



duration_months	credit_amount	purchase_purpose	job_type
	9	new car	
	9000	new car	

duration_months	credit_amount	purchase_purpose	job_type
41	3677	new car	unemp/unskilled non res
52	3919	new car	unemp/unskilled non res

Fig. 13. Interface for user study 2. Users can mark their preferences by clicking on the table.

data generated by NaturalSym. Given that columns such as longitude and latitude are intrinsically hard to distinguish, we believe that NaturalSym effectively preserves semantics in test generation.

4.4 User Study 2: User Preference

In user study 2, we place the data generated by BigTest and NaturalSym side by side for the same program path and allow users to mark their preferences. As shown in Fig. 13, we design a user study interface where participants can mark the box of the table to indicate their preferences. Participants can also select both tables if both appear realistic or choose neither if the content does not match the headers.

Nine PhD students participated in this user study. The results are also presented in Table 6. Participants marked 25.4 out of 28 tables generated by NaturalSym as realistic on average. Although several tables generated by NaturalSym are marked as unrealistic, all participants unanimously preferred the data generated by NaturalSym, underscoring its effectiveness.

4.5 Numerical Outliers

Numerical outliers are another common type of data unnaturalness. NaturalSym leverages the knowledge of statistical distribution to reduce outliers. For instance, if a program contains a branch predicate `elevation_ft > 1000`, symbolic execution would generate a value of 9000 for the true branch and 0 for the false branch. In contrast, NaturalSym leverages user annotations such as `Gaussian(`$\mu = 1442$`, `$\sigma = 1849$`)` to produce more typical values of 1469 and 447 for these branches, aligning closely with the mean of 1442 specified in the user-defined distribution.

To evaluate NaturalSym's effectiveness in reducing numerical outliers, we collect all numerical values from Gaussian columns in the tests generated by both NaturalSym and BigTest. Given that each column follows a unique Gaussian distribution, we normalize each value x into its absolute z-score, $\frac{|x-\mu|}{\sigma}$, to quantify the distance from the mean in terms of standard deviations. Fig. 14 displays the cumulative distribution of these z-scores. We use the 3σ-rule [40], a classical criterion for outlier detection, which identifies data points that lie more than 3σ away from the mean as outliers. Under this criterion, 44.7% of the data produced by BigTest are outliers, in contrast to only 0.95% from NaturalSym.

In addition, we conduct an ablation study by removing statistical distributions from user annotations, denoted as NaturalSym-N. Consider the scenario where a user annotates the movie rating column with the constraint $0 \leq$ `rating` ≤ 10 and a Gaussian distribution `Gaussian(`$\mu = 6.1, \sigma = 1.0$`)`.

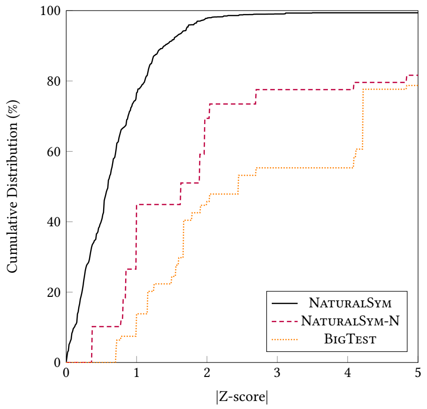


Fig. 14. Cumulative distribution of numerical values in Gaussian columns from NaturalSym, NaturalSym-N, and BigTest. We use the absolute z-score $\frac{|x-\mu|}{\sigma}$ as the x-axis for normalization, the cumulative probability $P(|\text{z-score}| \leq y)$ as the y-axis.

When we remove the distributional knowledge, NaturalSym-N retains only the range constraint $0 \leq$`rating`≤ 10 and consistently produces a value of 0 for this column. Overall, NaturalSym-N results in 22.2% fewer outliers than BigTest, yet it still produces 21.5% more outliers than NaturalSym. This demonstrates that NaturalSym can address constraint solvers' limitation in producing an extensive array of biased values by leveraging statistical distributional knowledge.

NaturalSym goes beyond merely pushing user-specified naturalness constraints into SMT solvers. While constraints such as `ivar ∈ Discrete(d1,d2)` can be effectively represented by the SMT formula `assert (or (= ivar d1) (= ivar d2))`, statistical distributional constraints like Gaussian distributions are not solvable by SMT solvers such as CVC5 [14] or Z3 [25]. Additionally, modern SMT solvers implement complex solving procedures, including simplex methods for linear programming. We investigated Z3 and CVC5 source code and found that integrating the sampling procedure within backend solvers is challenging. This finding justifies NaturalSym's pragmatic approach of handling sampling based on user annotations externally from the SMT solver implementation.

4.6 Comparison with NaturalFuzz

NaturalFuzz [36] is a coverage-guided fuzzer designed to produce natural tests for DISC programs. It profiles each row in the existing dataset against each program branch to measure its branch coverage and selects a small set of rows by maximizing branch coverage. Starting from this reduced set of rows, NaturalFuzz applies an interleaving mutation strategy to construct new rows by combining existing column values from different rows. As synthetic rows are built upon existing data, NaturalFuzz increases the chance of producing realistic tests. Hence, we select NaturalFuzz as a natural test generation baseline. Given that NaturalFuzz can generate new tests from existing ones, we further explore combining NaturalFuzz with the symbolic executor BigTest. Specifically, we evaluate three settings of NaturalFuzz: (1) NaturalFuzz as is, (2) NaturalFuzz-H (NaturalFuzz-Hybrid), taking seeds from both symbolic execution and original datasets, and (3) NaturalFuzz-P (NaturalFuzz-PureSymbolic), taking seeds from symbolic execution only.

In evaluation, we exclude three benchmarks for NaturalFuzz and NaturalFuzz-P, as well as six benchmarks for NaturalFuzz-H due to occurrences of crashes. NaturalFuzz can produce natural-looking tests like "338,250,US-OR,44.50,-123.29" for the airport dataset, where the third column is the airport code and the last two columns correspond to GPS location. As shown in Table

3, NaturalFuzz's perplexity score is 3.48, lower than NaturalSym's 4.90. NaturalFuzz produces natural tests due to its intrinsic nature of combining rows from existing datasets.

However, as shown in Table 4, NaturalFuzz, NaturalFuzz-H, and NaturalFuzz-P achieve 77.5%, 85.3%, and 76.2% lower path coverage compared to NaturalSym. As shown in Table 5, NaturalFuzz, NaturalFuzz-H and NaturalFuzz-P detect 97.9%, 94.6%, and 80.4% fewer injected faults compared to NaturalSym. NaturalSym outperforms NaturalFuzz for several reasons: (1) NaturalFuzz depends on seed tests to cover numerous program branches, but NaturalSym can symbolically generate new content absent in seed tests. (2) NaturalFuzz aims at improving statement coverage instead of path coverage. (3) Even after adding symbolic execution as a precursor, NaturalFuzz winnows out a small set of seeds based on branch coverage and removes valuable seeds from symbolic execution.

4.7 Comparison with TestMiner

TestMiner [27] predicts suitable values based on the target method signature. In its data mining phase, TestMiner collects test case calls such as *sqlParser.parse("SELECT x FROM y")* from Maven Central Repository [7] and indexes input literals *"SELECT x FROM y"* according to the test function signature *org.sql.SQLParser(java.lang.String)* into a retrieval model. In its retrieval phase, TestMiner looks up the retrieval model and provides values to test generators such as Randoop [49] based on the target method signature. For example, if we request TestMiner to provide inputs for *sqlParser.parse*, TestMiner will possibly return a valid SQL query string from its mined value pool rather than a random string such as "hi!" preset by Randoop. By mining from existing test calls and matching them with context words, TestMiner outperforms random test generators. Hence, we take TestMiner as another baseline for comparison.

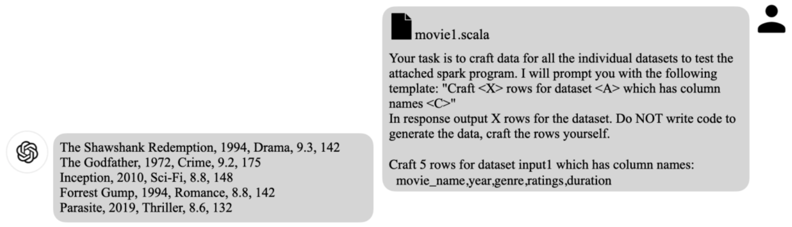
Since the Maven Central Repository lacks the structured dataset commonly required by DISC programs, we evaluate TestMiner in two settings: (1) TestMiner as is, and (2) TestMiner-DM+ (TestMiner-DataMining+) with additional data-mining from DISC program datasets. For TestMiner-DM+, we supplement its data mining phase with sampled rows from DISC program datasets. To generate tests, we query TestMiner using the signature of our target program, such as *NaturalSym.airport.test(java.lang.String).* For fairness, we require TestMiner and TestMiner-DM+ to generate the same amount of rows as NaturalSym.

As shown in Table 3, TestMiner achieves a perplexity score of 4.73 compared to BigTest's 40.01, suggesting TestMiner's producing natural-looking tests. However, the tests generated by TestMiner are of low quality for DISC programs. For instance, TestMiner generates a row "getAge" for the car_info dataset, whereas a row resembling "2432422,2014,28735,14788,2022-07-05,63342" is expected. Despite enhanced data mining efforts, TestMiner-DM+ still produces rows that do not pass the input validation stage of the target application.

As shown in Table 4, TestMiner and TestMiner-DM+ cover 96% and 75.1% fewer paths compared to NaturalSym. Table 5 shows that TestMiner detects no injected bugs while TestMiner-DM+ detects 98.7% fewer injected faults than NaturalSym. We find that (1) TestMiner is not effective in DISC program testing, as it is trained on other projects, and that (2) if we build indexes using a subset of data row samples, TestMiner effectively becomes a method of sampling from existing tests instead of synthesizing new tests.

4.8 A Case Study of GPT-4 on Natural Test Input Generation

To explore the feasibility of using Large Language Models (LLMs) for natural test generation, we conduct a case study using GPT-4 [8] to generate tests for DISC programs. We provide GPT-4 with the entire source code of the target function along with its input dataset schema. For a fair


movie1.scala

Your task is to craft data for all the individual datasets to test the attached spark program. I will prompt you with the following template: "Craft <X> rows for dataset <A> which has column names <C>"
In response output X rows for the dataset. Do NOT write code to generate the data, craft the rows yourself.

Craft 5 rows for dataset input1 which has column names:
movie_name,year,genre,ratings,duration

```
The Shawshank Redemption, 1994, Drama, 9.3, 142
The Godfather, 1972, Crime, 9.2, 175
Inception, 2010, Sci-Fi, 8.8, 148
Forrest Gump, 1994, Romance, 8.8, 142
Parasite, 2019, Thriller, 8.6, 132
```

Fig. 15. To generate tests using ChatGPT, we provide the Scala program under test and the schemas of the input datasets to ChatGPT. We explicitly request that ChatGPT produces the same number of rows as NaturalSym for fairness.

comparison, we explicitly instruct ChatGPT to generate the same number of rows as NaturalSym. Details of our prompting phrases are provided in Fig. 15.

ChatGPT can generate fairly natural-looking data, such as "The Shawshank Redemption,1994, Drama,9.3,142" in the movie dataset. Table 3 shows that ChatGPT's perplexity is 16.92, lower than BigTest's 40.01. However, its perplexity score is much higher than NaturalFuzz's 3.48 and NaturalSym's 4.90. This is because ChatGPT's generated data does not meet the input format requirement. For example, the movie duration column in the movie dataset is expected to be in the format of "02:22" (2 hours and 22 minutes), rather than simply stating the total minutes as "142". As shown in Table 4 and Table 5, ChatGPT only covers 25 program paths compared to NaturalSym's 110 and detects only 4 injected faults across 13 benchmarks compared to NaturalSym's 76.

Our empirical results indicate that (1) ChatGPT does not achieve adequate path coverage. Despite being provided with the complete target program, ChatGPT rather serves as a dictionary-based test generator based on the schema. (2) ChatGPT requires highly involved and detailed prompts. Even when provided with the input dataset schema, it still outputs synthetic rows that do not meet the requirements. Considering the highly structured nature of DISC program inputs, ChatGPT's performance in both path coverage and defect detection is inferior to that of NaturalSym. According to Wang et al. [64], none of the existing LLM-based approaches combine with constraint-based testing, like NaturalSym's capability of symbolic execution.

5 RELATED WORK

Natural testing. NaturalFuzz [36] extracts and combines column values from existing data to synthesize new rows for DISC testing. However, NaturalFuzz aims to improve statement coverage and does not enumerate path conditions like NaturalSym. TestMiner [27] leverages data mining techniques to predict suitable input string literals from existing test calls. Additionally, SDV [53] and Faker [5] are capable of generating natural-looking test cases. However, they require extensive manual configuration and often produce an overwhelming volume of test cases, making them less effective for DISC testing [36]. Houkjær et al. [33] propose a data generation framework incorporating a graph-based approach, allowing users to provide data types and distributions to enhance data naturalness. DGL [17] is a framework to generate large-scale datasets with intra-table and inter-table correlations. DOMINO [10] focuses on increasing schema coverage in dataset generation. In comparison, NaturalSym is a symbolic execution-based tool that targets high path coverage and high fault detection while maintaining the naturalness of the generated inputs.

DISC application testing. BigTest [30] is a symbolic execution-based testing for DISC applications but cannot generate natural-looking tests like NaturalSym. Various fuzzers [36, 71] use framework abstraction to efficiently fuzz DISC applications without invoking the distributed system. Csallner et al. [23] are the first to use dynamic symbolic execution to generate test inputs for MapReduce programs. This technique mainly identifies non-determinism bugs arising from the violation of UDF correctness conditions. Xu et al. [67] further develop the technique by checking additional properties such as statefulness and blocking.

Dictionary-based fuzzing. Dictionary-based mutation was introduced [70] to compensate for the grammar blind nature of AFL. However, dictionary-based fuzzing is suboptimal in our scenario. Given the highly structured nature of the input datasets, dictionary-based testing often fails to pass input validation and struggles to detect deep semantic bugs. In contrast, NaturalSym produces sample sets based on user annotations as a guide for naturalness, while aiming to improve path coverage using constraint solvers.

Symbolic execution. Previous studies [13, 38, 63] leveraged user-specified preconditions to compensate for information that symbolic executors cannot capture from branch predicates. These preconditions typically include the length of a buffer or the prefix of a buffer, primarily employed to mitigate path explosion in symbolic execution. NaturalSym has expanded the types of user annotations. Moreover, it perceives user specifications as soft guidance rather than hard constraints, aiming to maximize path coverage.

Property-based testing. Property-based testing (PBT) is a powerful testing technique popularized by QuickCheck [22] in Haskell. PBT uses executable properties to generate tests and has successfully discovered numerous in a range of real-world softwares [11, 12, 34, 35]. Several works [39, 43, 50] have enhanced property-based testing with coverage guidance. Recently, Vikram et al. [62] provide LLMs with additional API documentation to generate property-based tests. The application of PBT to DISC testing continues to be a promising area for exploration.

LLM-based testing. Large language models are proven to be successful over many tasks when properly prompted [16, 59]. Many techniques [9, 56–58, 60] pre-train or fine-tune LLMs for unit test generation. Several approaches [24, 29, 42, 54, 62, 66, 68, 72] design effective prompting words for unit test generation. Multiple works [44, 46, 47, 61] utilize LLMs for test oracle generation. Lemieux et al. propose CodaMosa [41], which engages LLMs when the coverage of search-based testing plateaus. However, according to Wang et al. [64], there is no such LLM-driven test generation method that integrates with constraint-based testing.

6 CONCLUSION

We are the first to incorporate the notion of naturalness into symbolic execution-based test generation without sacrificing path coverage. We instantiated the idea in NaturalSym, a novel symbolic test generation tool. NaturalSym enables users to specify the underlying semantics of program inputs, capturing information that symbolic execution cannot. Based on user annotations, it can draw realistic examples for input variables and prioritize their use during constraint solving. NaturalSym significantly enhances test naturalness and detects 17.4 more injected faults on average than the state-of-the-art DISC symbolic executor BigTest, by only adding 1.47× runtime overhead. NaturalSym also outperforms alternative natural test generators in terms of path coverage and defect detection, finding injected faults 47.8× more than NaturalFuzz and 19.1× more than ChatGPT, while TestMiner fails to detect any injected faults.

DATA-AVAILABILITY STATEMENT

All scripts and datasets are available on Zenodo [65].

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant numbers 2106838, 1764077, 1956322, 2106404, and 2106420. It is also supported in part by funding from Amazon and Samsung. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

REFERENCES

[1] Accessed: 2023. Airport Codes Dataset. https://datahub.io/core/airport-codes#resource-airport-codes

[2] Accessed: 2023. Carvana Car Sales Dataset. https://aws.amazon.com/marketplace/pp/prodview-y77x3t6zisn4w

[3] Accessed: 2023. ChatGPT. https://openai.com/blog/chatgpt

[4] Accessed: 2023. DistilGPT2. https://huggingface.co/distilgpt2

[5] Accessed: 2023. Faker. https://faker.readthedocs.io/en/master/

[6] Accessed: 2023. IMDB Dataset. https://www.imdb.com/search/title/?title_type=feature,tv_movie&ref_=adv_prv'

[7] Accessed: 2024. Maven Central. https://central.sonatype.com

[8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). https://doi.org/10.48550/arXiv.2303.08774

[9] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023). https://doi.org/10.48550/arXiv.2302.10352

[10] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. 2018. DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 12–22. https://doi.org/10.1109/ICST.2018.00012

[11] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2–10. https://doi.org/10.1145/1159789.1159792

[12] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[13] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84. https://doi.org/10.1145/2560217.2560219

[14] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[15] 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, UK)*, Vol. 13. 14.

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901. https://dl.acm.org/doi/abs/10.5555/3495724.3495883

[17] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators *(VLDB '05)*. VLDB Endowment, 1097–1107. https://dl.acm.org/doi/10.5555/1083592.1083719

[18] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[19] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38. https://doi.org/10.1145/1455518.1455522

[20] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394. https://doi.org/10.1109/SP.2012.31

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–49. https://doi.org/10.1145/2110356.2110358

[22] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279. https://doi.org/10.1145/351240.351266

[23] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. 2011. New ideas track: testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 504–507. https://doi.org/10.1145/2025113.2025204

[24] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468. https://doi.org/10.1016/j.infsof.2024.107468

[25] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[26] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113. https://doi.org/10.1145/1327452.1327492

[27] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. 2017. Saying 'hi!'is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 44–49. https://doi.org/10.1109/ASE.2017.8115617

[28] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing. In *NDSS*, Vol. 8. 151–166.

[29] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. 15–24. https://doi.org/10.1145/3624032.3624035

[30] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-Box Testing of Big Data Analytics with Complex User-Defined Functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953

[31] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. Bigsift: automated debugging of big data analytics in data-intensive scalable computing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 863–866. https://doi.org/10.1145/3236024.3264586

[32] Hans Hofmann. 1994. Statlog (German Credit Data). UCI Machine Learning Repository. https://doi.org/10.24432/C5NC77

[33] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1243–1246. https://dl.acm.org/doi/10.5555/1182635.1164254

[34] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[35] John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 135–145. https://doi.org/10.1109/ICST.2016.37

[36] Ahmad Humayun, Yaoxuan Wu, Miryung Kim, and Muhammad Ali Gulzar. 2023. NaturalFuzz: Natural Input Generation for Big Data Analytics. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, San Francisco, CA, USA. https://doi.org/10.1109/ASE56229.2023.00034

[37] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678. https://doi.org/10.1109/TSE.2010.62

[38] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 553–568. https://doi.org/10.1007/3-540-36577-X_40

[39] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360607

[40] Rüdiger Lehmann. 2013. 3 σ-rule for outlier detection from the viewpoint of geodetic adjustment. *Journal of Surveying Engineering* 139, 4 (2013), 157–165. https://doi.org/10.1061/(ASCE)SU.1943-5428.0000112

[41] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*. https://doi.org/10.1109/ICSE48619.2023.00085

[42] Vincent Li and Nick Doiron. 2023. Prompting code interpreter to write better unit tests on quixbugs functions. *arXiv preprint arXiv:2310.00483* (2023). https://doi.org/10.48550/arXiv.2310.00483

[43] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://doi.org/10.21105/joss.01891

[44] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1580–1598. https://doi.org/10.1109/TSE.2022.3183297

[45] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB*, Vol. 6. 1049–1058.

[46] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. https://doi.org/10.1109/ICSE48619.2023.00205

[47] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2111–2123. https://doi.org/10.1109/ICSE48619.2023.00178

[48] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*. Springer, 156–169. https://doi.org/10.1007/11814948_18

[49] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84. https://doi.org/10.1109/ICSE.2007.37

[50] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401. https://doi.org/10.1145/3293882.3339002

[51] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 179–180. https://doi.org/10.1145/1858996.1859035

[52] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20 (2013), 391–425. https://doi.org/10.1007/s10515-013-0122-2

[53] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic Data Vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 399–410. https://doi.org/10.1109/DSAA.2016.49

[54] Laura Plein, Wendkûuni C Ouédraogo, Jacques Klein, and Tegawendé F Bissyandé. 2023. Automatic generation of test cases based on bug reports: a feasibility study with large language models. *arXiv preprint arXiv:2310.06320* (2023). https://doi.org/10.48550/arXiv.2310.06320

[55] Nihar Ranjan, Kaushal Mundada, Kunal Phaltane, and Saim Ahmad. 2016. A Survey on Techniques in NLP. *International Journal of Computer Applications* 134, 8 (2016), 6–9. https://doi.org/10.5120/ijca2016907355

[56] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 409–420. https://doi.org/10.1109/ASE56229.2023.00193

[57] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2023. Domain Adaptation for Deep Unit Test Case Generation. *arXiv e-prints* (2023), arXiv–2308. https://doi.org/10.48550/arXiv.2308.08033

[58] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. *arXiv preprint arXiv:2310.02368* (2023). https://doi.org/10.48550/arXiv.2310.02368

[59] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023). https://doi.org/10.48550/arXiv.2302.13971

[60] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020). https://doi.org/10.48550/arXiv.2009.05617

[61] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64. https://doi.org/10.1145/3524481.3527220

[62] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023). https://doi.org/10.48550/arXiv.2307.04346

[63] Willem Visser, Corina S Pv asv areanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. 97–107. https://doi.org/10.1145/1007512.1007526

[64] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024). https://doi.org/10.1109/TSE.2024.3368208

[65] Yaoxuan Wu, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2024. Reproduction Package of 'Natural Symbolic Execution-based Testing for Big Data Analytics'. https://doi.org/10.5281/zenodo.11090237

[66] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023). https://doi.org/10.48550/arXiv.2305.04764

[67] Zhihong Xu, Martin Hirzel, Gregg Rothermel, and Kun-Lung Wu. 2013. Testing properties of dataflow program operators. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 103–113. https://doi.org/10.1109/ASE.2013.6693071

[68] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023). https://doi.org/10.48550/arXiv.2305.04207

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. https://dl.acm.org/doi/10.5555/1863103.1863113

[70] Michał Zalewski. 2015. afl-fuzz: making up grammar with a dictionary in hand. (2015). https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

[71] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 722–733. https://doi.org/10.1145/3324884.3416641

[72] Ying Zhang, Wenjia Song, Zhengjie Ji, Na Meng, et al. 2023. How well does LLM generate security tests? *arXiv preprint arXiv:2310.00710* (2023). https://doi.org/10.48550/arXiv.2310.00710

Received 2023-09-28; accepted 2024-04-16