Self-Verifying Data

Douglas Hoffman
Software Quality Methods, LLC.
Doug.Hoffman@acm.org

Abstract

Some tests require large data sets. The data may be database records, financial information, communications data packets, or a host of others. The data may be used directly as input for a test or it may be pre-populated data as background records. Self-verifying data (SVD) is a powerful approach to generating large volumes of information in a way that can be checked for integrity. This paper describes three methods for generating SVD, two of which can be easily used for large data sets.

For example, a test may have a prerequisite that the database contains 10,000,000 customers and 100,000,000 sales orders. The test might check adding new customers and orders into the existing data, but not directly reference the preset data at all. How can we generate that kind of volume of data and still be able to check whether adding customers and orders might erroneously modify existing records? SVD is a powerful, proven approach to facilitate such checks.

The paper and talk describe the concepts, applications, and methods for generating such data and checking for data corruption. They cover:

What self-verifying data is
Why and how self-verifying data can be used
Applications where such data is useful
Three ways to apply self-verifying data
How to check the data records generated this way

Biography

Douglas Hoffman BACS, MSEE, MBA, ASQ Fellow, ASQ-CSQE, ASQ-CMQ/OE

Doug has more than thirty years of experience with software engineering and quality assurance. Today he teaches and does management consulting in strategic and tactical planning for software quality. Training specialties include context-driven software testing, test oracles, and test automation design. His technical specialties include test oracles, test planning, automation planning, and developing test architectures. Management specialties include ROI based planning of software engineering mechanisms, QA management, organizational assessment, evaluating and planning for organizational change, managing and creating project management offices, building new quality organizations, and transforming existing quality assurance and testing groups to fulfill corporate visions.

Doug is very active professionally: President of the professional society Association for Software Testing, ASQ Fellow, Senior member of ACM and IEEE, past Chair of the Silicon Valley Section of ASQ, past Chair of the Silicon Valley Software Quality Association, Chair and organizer of numerous quality conferences, over 50 conference presentations, and written dozens of publications on software testing, quality assurance, and management. He holds degrees in Computer Science, Electrical Engineering, and an MBA. He also has ASQ certifications in Software Quality Engineering and Management of Quality/Organizational Excellence.

Introduction

One view of running tests is observing the software under test (SUT) for proper behavior given some stimulus. The stimulus comes from many sources, direct input and referenced data among them. The amount and nature of the required data varies based upon the requirements of the software under test and the purpose of the test. Although not always the case, good tests should generally be repeatable, start from a known state, and have verifiable results (performance tests, race condition tests, and multi-threaded asynchronous tests being some counter examples). Therefore, these tests require data that is repeatable and will result in verifiable outcomes when processed by the SUT.

Tests sometimes require large volumes of inputs (e.g., run-time generation of millions or billions of values) and huge data sets (e.g., pre-populated databases) that can result in difficulties in verification of test outcomes and determining of test verdicts. Self-verifying data (SVD) can answer questions like “did any other records get corrupted?” when a test is complete. Detection of reasonable or unexpected software behavior is an oracle problem that is compounded with large volumes of data. Self-verifying-data is self-descriptive data, i.e., the data contains the key or clues as to what the data is supposed to be. For example, “This sentence is 52 characters in length (including spaces).”

SVD is not useful for all test situations. Sometimes the SUT has no requirement for pre-populated data (e.g., computations in a calculator) or there may be no place for a key (e.g., uid/password combinations). The SUT’s processing of data may be too complex to be determined with a single key (e.g., image analysis). But, SVD can be a powerful way to tag and track expected outcome checking.

Although data generation and outcome verification may be integrated within a test, for simplicity in this paper I will logically separate data collection/generation from running the test on the SUT from the verification of outcomes. Whether the data is generated before or during a test is not important to the concept of SVD, nor is it important whether the verification of outcomes occurs during or after the SUT is run. Logically separating data generation, SUT execution, and verification allows us to examine SVD more clearly.

Some notation and terminology:

CRC	A Cyclic Redundancy Check is a computed numeric value that represents a set of data. Any changes in the data will result in computing a different CRC value
Oracle	A test oracle is the principle or mechanism by which we recognize potential erroneous SUT behavior (e.g., comparison with expected results)
RAND	A random number
RNG	Random number generator
Seed	A number used to initialize a RNG
SUT	Software under test
SVD	Self-Verifying Data

1.1 Data generation

Data for SVD can be generated by a human or by a computer. Generally speaking, once the concept of SVD is understood, a human can be very creative in its application. However, the volume of human generated data is severely limited in comparison with computer generated data (unless you consider the human writing programs to generate the data). In this paper I lean heavily toward computer generated data that is useful in interesting automated tests.

Actual generation of well-formed data is generally straightforward. The rules for a grammar can be defined and data generated following the rules. The trick in SVD is embedding a key with the generated data for later verification. The key does not have to be unique but it needs to have very high probability of matching the data to be very useful. For example, a computed 32 bit cyclic redundancy check (a.k.a. CRC) always matches with the data used to generate it but will randomly match other data 0.000000025% of the time.)

1.1.1 Random numbers

Randomly generated data is a useful mechanism for augmenting the specific test values needed for testing. Boundary conditions and other special cases provide the foundation for functional testing, while random values can provide tests for hidden special cases or generation of filler data. There are several ways to select or generate large volumes of test data. When we generate high volumes of SVD we generally employ random number generators (RNG) to select or construct the data. There are a few important characteristics of computer random numbers that are particularly relevant for this process.

- Computer random numbers are *pseudo*-random numbers. They exhibit statistical randomness but are generated using repeatable algorithms. I.e., a large set of random numbers have the statistical characteristics of truly random numbers, but the specific values form a repeating sequence that cycles after an astronomical number of generated values.

- If an initial value (called a *Seed* value) for the random sequence is specified, the same sequence of random numbers will be generated. If no *Seed* value is given, the RNG chooses one using highly unpredictable sources (like the system clock).

Using a seed value we can repeat a series of random numbers. This means we can reliably rerun a test or data generation that uses random numbers.[1] If we want a new random sequence we can use the random number generator to create a new *Seed* value. The RUBY code in Figure 1 shows an example of code using repeatable random sequences.

```
MAX_SEED = 1_000_000_000

 def initial_RNG_seed(myseed)

    if (myseed == nil) # Check if seed value is provided
       # Create a random number to seed RNG
       Puts "(no seed passed in, so generate one)"
       myseed = srand() # start the random sequence randomly
       myseed = rand(MAX_SEED)  # generate a new Seed value
    end

    # Save the seed so that we KNOW what seed we used
    puts "myseed is #{myseed.to_s}\n"
    foo2 = srand(myseed) # initialize the RNG
    foo = rand() # generate the [first] random number
    return foo
end
```

Figure 1: Ruby code for generating or reusing random number seeds

[1] Some care is required for reliable repetition or random sequences. For example, asynchronous threads and programmatic variable scope changes are difficult to manage.

Figure 2 shows sample output of the seed and first random value from invoking the initialization routine in Figure 1. If a *Seed* value is provided it is used to initiate the same random sequence (shown in the second and fifth cases). If no *Seed* value is provided the program generates and logs a new randomly generated *Seed*.

```
puts ("First run: #{initial_RNG_seed(nil)} \n \n")
puts ("Second run: #{initial_RNG_seed(400)} \n \n")
puts ("Third run: #{initial_RNG_seed(nil)} \n")

→

(no seed passed in, so generate one)
myseed is 144288918
First run: 0.3705579466087263

myseed is 400
Second run: 0.6687289088341747

(no seed passed in, so generate one)
myseed is 108495905
Third run: 0.09838898989988143

puts ("Fourth run: #{initial_RNG_seed(nil)} \n \n")
puts ("Fifth run: #{initial_RNG_seed(400)} \n \n")
puts ("Sixth run: #{initial_RNG_seed(nil)} \n")

→

(no seed passed in, so generate one)
myseed is 173770139
Fourth run: 0.5201072369109297

myseed is 400
Fifth run: 0.6687289088341747

(no seed passed in, so generate one)
myseed is 320781144
Sixth run: 0.020652681349145108
```

Figure 2: Example output from running the Ruby code in Figure 1

1.1.2 Timing of record generation

The test data may be generated independently before the SUT is exercised or as part of the test. Populating large data sets is done before running the test. Randomly generating data communication packets as input to the SUT is done within the test. Although designed and applied somewhat differently, the timing of data generation is unimportant for the purpose of SVD, since the SVD mechanisms are the same either way.

1.2 Data verification

The purpose of generating SVD is so the data itself contains clues as to what the data should be. As described below, there are several SVD approaches and techniques. Though the oracle techniques applicable to each type of SVD are somewhat different, the common thread is that the data provides information for verification of test outcomes.

2 Three mechanisms for SVD

SVD can be generated in three ways: self-descriptive, cyclic algorithms, and random generation. Each is described below.

2.1 Self-descriptive data

We can create data that describes its own attributes. 20 point text, there is a box around these characters, and *underlined italicized Arial text* are some examples. This approach is primarily used in human (manual) testing although it is possible to programmatically generate such data. For self-descriptive data the interpretation of the data becomes the source for the oracle, which is extremely difficult to automate, as discussed in Section 3.1.

2.2 Cyclic data

Test data may be generated using a pattern of repetition. An initial value is created and additional values concatenated based on duplication or a simple repetitive algorithm. The key(s) are prepended or appended to the data. As described in Section 3.2, the oracle is then implemented by repeating the data generation from the key and comparing data, or by identifying the pattern and thus deriving and comparing the key(s).

2.2.1 Repeated values

The simplest method is generating text or arithmetic values by repeatedly concatenating data. The pattern key can be fully defined with two values: the starting value and number of iterations. This works equally well for text or numeric data.

For example, “ab10abababababababababab” and {55, 10, 55, 55, 55, 55, 55, 55, 55, 55, 55, 55} are repeating series. Cyclic data of this type usually has the key prepended when a generation oracle is applied and added at the end when pattern recognition is used. (See Oracle mechanisms in Section 3.2.)

Some care is required with text patterns to avoid ambiguous patterns, especially with manually generated sequences. Although highly unlikely with randomly generated data, the possibility exists. The data portion of “ab10abababababababababab” is the same as “abab5abababababababababab” and “abababababababababab1abababababababababab,” so there are three different ways this pattern could be interpreted and a key derived from the generated data.

2.2.2 Extended numeric sequence

An extended numeric sequence is a series of different numbers based on very simple algorithms. The algorithm generates each sequential value using a simple transformation of the previous value. Addition of a constant is the most common method. The pattern key can be fully defined with three values: starting value, constant, and count of numbers.[2]

One example of an extended numeric sequence is a numeric sequence where each number is the result of adding a constant to the previous number ($X_n = X_{n-1} + K$, where K is a constant)[3]. The set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68} is an example where the key is {5, 7, 10} (start with 5 and add 7 for 10 values) and {5, 12, 19, 26, 33, 40, 47, 54, 61, 68} are the values. With the numeric values concatenated with the key values, checking can be done using either oracle method described in Section 3.2.

2.3 Random data

Probably the most powerful mechanism for generating SVD is using random numbers. Although self-descriptive and cyclic data can be generated randomly, random SVD depends only on the use of the same RNG and Seed values. The key used in random SVD is the Seed value. The seed is either embedded within the data or otherwise associated with the records as described in Sections 2.3.4 and 2.3.5.

Random SVD can be generated and used in several ways as described in Sections 2.3.1 through 2.3.3.

2.3.1 Nonsense padding

For this case data is collected or generated for space-filling purposes without expectation that any interpretation or processing of the records will occur. The values generated are place holders used to increase the amount of data and are passive for the purposes of the test(s). The SVD in this case is used as verification that the data has not become corrupted. This type of SVD is often used for performance and stress tests.

Because the data is not specifically processed, the values do not need to be meaningful. It only needs to be well formed enough to not throw exceptions. For example, names and addresses do not need to be meaningful, but only acceptable characters should be in the fields. The address, for example, does not need to have a real city’s name.

For example, we may generate a million random users to assess an application with a large user base. These users are added to the set of specific users needed by the basic functional tests. None of the randomly generated users are accessed because each test uses their own specific users. Yet, all tests may include the larger data set for robustness.

2.3.2 Well-formed random records

For this case data is generated in well-formed records that may or may not be processed. A grammar is created describing the record and it is then applied using random values. Figure 3 shows a RUBY program to generate simple well-formed arithmetic statements using a hard-coded grammar.[4] (The generated statements are randomly created without seed values for simplicity.) Note that separate grammars may be required for each data field and data constraints among fields can make generation very complex.

[2] The cyclic numeric and text SVD mechanisms were used to test randomly generated data packets sent across a multi-modal network that included some half-duplex links (Ethernet, radio, power lines, and four other types of networks in series) where returning the data for analysis was impractical.
[3] Note that numeric overflows may occur, in which case the value can be truncated and the sequence continued.
[4] Although difficult to achieve, this type of generation of well-formed data can be generalized by reading in a grammar specification, usually for some group of data formats. For example, specifying different math functions or data base records.

```
def eq_gen
    jubilee = 1 + rand(8) # Matches the 4 cases we have defined
    case jubilee
      when 1
          "(" << eq_gen() << ") + (" << eq_gen() << ")"
      when 2
         "(" << eq_gen() << ") - (" << eq_gen() << ")"
      when 3
        "(" << eq_gen() << ") * (" << eq_gen() << ")"
      when 4
         "(" << eq_gen() << ") / (" << eq_gen() << ")"
      else
        rand(100).to_s  # 50% chance of generating a number
    end
end
```

Figure 3: Ruby code to generate arithmetic expressions

Figure 4 provides some examples of random well-formed equations generated by the program.

```
puts ("First:  #{eq_gen.to_s} \n\n")
puts ("Second:  #{eq_gen.to_s} \n\n")
puts ("Third:  #{eq_gen.to_s} \n\n")
puts ("Fourth:  #{eq_gen.to_s} \n")


→

First:  (77) - ((62) / (6))

Second:  ((10) - (40)) + (67)

Third:  53

Fourth:  (62) - ((96) * ((((77) - (72)) - ((7) * ((47) - (91)))) / ((34) + (((70) - (18)) + (4)))))
```

Figure 4: Randomly generated expressions from Ruby code in Figure 3

2.3.3 Execution time random data generation

One useful way to use SVD is generating test values as part of a test. Either nonsense or well-formed SVD can be generated by the test as input to the SUT.[5] Nonsense data is likely to trigger error mechanisms in the SUT (which may be the intent of the test). Well-formed data is more likely to be processed by the SUT and the results of processing can be checked.

[5] This technique may be applied without appending the key, depending on the oracle mechanisms used. An example of execution time generation of well-formed input without a key is function equivalence testing (where the generated input is sent to the SUT and an alternate implementation such as different spreadsheets) and the returned results compared within the test. Since this does not require SVD, there is no further discussion of this mechanism.

The results from the SUT processing may be checked as the test progresses or through post processing the data. Checking as the test progresses may be done within the test or by a separate oracle mechanism receiving or monitoring the results.

2.3.4 Appending a seed value

Sometimes it is possible to simply append the seed to the test data. For example, the seed could be the first in a numeric sequence of random numbers. In the example below, the seed is appended to the randomly generated text.

For the example shown in Figure 5:

- Assume the seed (S) is 8 characters and name field holds a maximum of 128 characters
- To generate a random name:
  1. Generate a random number seed (S) and convert it to a string (assume 8 characters)
  2. Initialize the random number generator using SRAND(S) (or RAND(S))
  3. Generate a random Length (L) (up to 120 characters) using RAND()
  4. Generate a random name (N) with L characters using RAND()
  5. Concatenate the seed to the name
- Verify the name by extracting the seed and regenerating it

Name = | … *L* Random characters | 8 character Seed |

Figure 5: An example of embedding the seed within test data

Another example from Section 2.2.2 is the set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. Here the key is embedded as the first three values describing the cyclic series {5, 7, 10}, followed by the generated data {5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. The ten data values can be easily generated starting with 5 and adding 7 each time. [3]

2.3.5 Adding a field for the seed value

Sometimes it is not possible to append the seed with the data. For example, date fields and binary values have no flexibility for added information. In these instances we need to add an associated field with the seed value. This is straightforward for database records by adding a new field, but can be tricky with data in say, a checkbook application.

For example:

To create a database record:
1. Generate a random number Seed (S)
2. Store the Seed value in an added field within the record
3. Generate the record using the Seed and an algorithm
4. Regenerate the records by applying the algorithm with each Seed

As mentioned, some data sets do not lend themselves to easily appending the seed values. Often there are unused fields or free form fields for notes where the seed can be inserted. Where there is no available field it may be necessary to create a separate table to contain the seeds. Each seed needs to be associated with its generated data, so some unique record identifier is needed. For example, in a checkbook application we might create a table linking the check numbers or deposit amounts/dates with their corresponding seeds.

3 Oracle Mechanisms for SVD

As described in the introduction, the oracle is how we tell good SUT behavior from problematic behavior. An oracle can be built into the test, where the test periodically checks for expected results. It can also be built separately from the test, to be run during or after running the test. There is no limit on the number of oracles that can be created to check specific test outcomes or general program and system attributes. (A memory leak detector is an example of a separate check that runs in parallel with the regular tests.) The focus here is on the specific oracle mechanisms for SVD.

Depending on the application of the SVD, verification can happen during or after running the test (or both). SVD is typically not required when results are checked within a test because the expected outcomes being checked do not need to be reconstructed using a key. An advantage of execution-time checking is possibility of stopping to diagnose errors when they occur. Post-execution checking is a matter of reading the SVD and using the key to verify it. An advantage of post execution checking is exposing bugs for conditions we did not consider when the test was designed and implemented. However, post-execution checking can only identify that data was corrupted.

For example, the checkbook records can each be rechecked after test completion to detect unintended side effects. Where the test may check for specific results and expected potential side effects, there are an infinite number of ways the SUT may unexpectedly modify other data. Post execution checks are likely to uncover corrupted data from any other bugs or side effects not specifically checked in a test.

3.1 Oracle mechanisms for self-descriptive data

Self-descriptive data is usually hand-generated and uses a human as the oracle. Automatic generation of self-descriptive data is difficult, but can be achieved using the random well-formed data approach described in Section 2.3.2. Automating an oracle for randomly generated, self-descriptive data is very difficult because a key only provides a syntactical description of the data. Recognizing the actual attribute(s) of the data is a sematic problem. A human tester can read and interpret this kind of information more flexibly and reliably than a computer. However, checking whether the data has the specified attributes can be straightforward if they can be recognized.

For example, "20 point text" is self-descriptive and the size attribute can be randomly chosen. However, automating the recognition that it *is* 20 point text is nearly impossible, while a human can easily distinguish between that and "20 point text."

3.2 Oracle mechanisms for cyclic data

There are two applicable oracle mechanisms for cyclic data: regeneration of the data and pattern identification. Either mechanism can be run during a test or afterwards.

Regeneration of the data is done by applying the same generation rules using the key values. The resulting data can then be compared. For large data sets, generation and comparison may be run without saving the generated data and in some situations it may be possible to generate and compare without keeping either the test or oracle data by generating and comparing expected results as the test runs.

Pattern identification is done by deriving the key value(s) by analysis of the data and then comparing keys. In the case of a data communications test the oracle was at one end of the chain of links the test was run at the other end. The test generated data was verified and the pass/fail result returned. For the set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68}, the first data value is 5, the difference between values is 7, and there are 10 data values {5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. Thus, the key is {5, 7, 10} (which matches the SVD key).

3.3 Oracle mechanisms for random data

The most practical test oracle is to apply the original data generation algorithm using the seed to regenerate the expected data. (Reverse engineering the seed is basically a decryption task with very little data for each seed, which is both time consuming and error prone.) The oracle can be as simple as reading the row from a table and using the key to regenerate the data for comparison, or quite complex; for example extracting the key from a note field to check the data extracted from a printed invoice for a randomly created sales order . The oracle usually uses the same program (or a variation of the program) to regenerate the data since the algorithm is established to generate that very data.

Post-processing is typical for randomly generated SVD, although there are cases where the expected results are checked as the test runs.[6] Post-processing has the advantage of checking all the records, while checking during a test run can catch problems early to make diagnosis easier. An independent oracle may also check SVD records continuously while tests are running (e.g., checking and rechecking database records).

4 Conclusions

SVD embeds a key within a set of data that describes the data itself. SVD is very useful for testing in many contexts because much of the software we test is stimulated using data that can be self-descriptive, and therefore checked for side effects. SVD can be self-descriptive, cyclic, or random. Self-descriptive SVD is usually generated and checked by humans. Cyclic SVD employs a repeating pattern that can be described with a few key values. Random SVD uses random numbers to create and tag test data. The trick with checking data is recognizing when the SUT does something unexpected (the oracle problem). SVD is one approach that facilitates checking outcomes because the key to the expected result is embedded within the data.

- SVD may be useful when:
  - The data lends itself to self-description
  - The data can be thought of as records
  - A key or seed can be used for data generation
  - Incorporation of the key with the data is straightforward
  - The test requires a high volume of inputs or referenced data
  - Checking for data corruption after test completion

- Not useful when:
  - The SUT does not use record type data
  - Checking data within a test
  - Outcomes don't reflect SVD type data records
  - The data structure is too complex
  - The data is not easily generated from a key
  - The key is not easy to include with the data

[6] Randomly generated data was also used in the data communications testing described in Footnote 2.